

## ABSTRACT

This project is about VLSI floorplanning optimization. Floorplanning optimization is used to minimize the deadspace of the floorplan. This is to reduce cost for die fabrication, minimize resistance in the circuit and also reduce heat produced. Hence, VLSI floorplanning is important in IC design. Floorplanning optimization consists of representation and optimization algorithm. In present work, Dot Model (DM) and Corner Bottom Left List (CBLL) were developed as floorplan representation. These two models are based on topological placement method. DM is optimized using genetic algorithm (GA). GA is a widely used optimization algorithm based on the concept of survival of the fittest. This means that a population with random generated sequence will be generated and the fitness of the population will be evaluated. The best quantile of the population will be maintained and genetic operations will be performed on these chromosomes. The selected best quantile population will be brought to the next generation. GA is able use the representation for DM by modifying the chromosomes to match the tuples for DM for optimization. Two methods of optimization are used for CBLL. They are Cross Entropy and also Genetic Algorithm. CE is a new algorithm that was recently developed using probability. This method consists of 2 phases which are the random data generation and then update of the probabilities based on the performance of the data generated. This method is used to reduce the stochastic of data generation as the second iteration will have influence of the first iteration data. The generation of strings are based on three dimensional matrices to obtain the probability between each block to another block. These algorithms are tested on MCNC benchmarks which are apte, xerox, hp, ami33 and ami49. DM-GA gives fair results of deadspace for the benchmarks tested. However, DM uses a long runtime to decode the floorplan. CBLL- GA has shorter optimization runtime compared to DM-GA because CBLL can decode the string much faster. Both methods give almost similar deadspace

area. CBLL-CE gives the least deadspace area. CE is able to calculate and give the relationship of the local deadspace area during placement and determine the best combination between the adjacent blocks. However, CE requires longer run time compared to GA because the parameters of the random mechanism need to be updated in each iteration.

## ABSTRAK

Projek ini adalah mengenai pengoptimuman pelan lantai VLSI. Pengoptimuman pelan lantai VLSI adalah untuk meminimumkan ruang kosong kawasan pelan lantai VLSI. Ini adalah untuk menurunkan kos untuk memfabrikkan die, mengurangkan kerintangan dalam litar dan juga mengurangkan penghasilan haba. Ini menunjukkan kepentingan dalam mengoptimumkan pelan lantai VLSI. Pengoptimuman pelan lantai VLSI terbahagi kepada dua bahagian iaitu perwakilan modal dan juga pengoptimuman. Dalam kajian ini, Modal Bintik (Dot Model – DM) dan Senarai Bahagian Bawah Kiri (Corner Bottom Left List – CBLL) digunakan untuk perwakilan modal. Kedua-dua kaedah ini menggunakan kaedah perletakkan topologi. Kaedah pengoptimuman yang digunakan bersama DM adalah Algoritma Genetik (Genetic Algorithm – GA). GA digunakan secara meluas sebagai kaedah optimum dengan menggunakan populasi kromosom dan menggunakan konsep penyesuaian hidup yang terbaik. Satu populasi akan dijanakan secara rawak. Kemampuan setiap gen dalam populasi tersebut akan dinilai. Selepas itu, 20% yang terbaik akan disimpan di dalam populasi tersebut dan yang selainnya akan diabaikan. Selepas itu, operasi mutasi dan penyilangan akan dibuat kepada populasi tersebut. Sekiranya anak gen adalah lebih lemah daripada ibu gen, gen tersebut akan dibuang. Kromosom GA diubahsuai mengikut perwakilan DM agar optimasi dapat dibuat. Dua kaedah optimasi digunakan untuk CBLL iaitu GA dan entropi persilangan (Cross Entropy – CE). CE merupakan algoritma yang baru dihasilkan menggunakan kaedah kebarangkalian. Kaedah ini terdiri daripada dua fasa iaitu penjanaan data secara rawak dan kemaskini kebarangkalian berdasarkan prestasi data yang dijanakan. Kemaskini kebarangkalian adalah untuk mengurangkan data dan data dalam iterasi pertama akan mempengaruhi data iterasi kedua. Penjanaan data dibuat menggunakan matriks tiga dimensi untuk memperolehi perkaitan antara modal-modal. Semua algoritma ini diuji dengan tanda MCNC iaitu apte, xerox, hp, ami33 dan ami49. DM

dan GA memberi keputusan ruang kosong yang serdahana baik. Akan tetapi, DM menggunakan masa yang lebih panjang untuk dinyakod berbanding dengan CBLL. Oleh itu, CBLL dan GA mengambil masa yang lebih pendek berbanding dengan DM dan GA. Kedua-dua kaedah ini memperolehi ruang mati yang lebih kurang sama. CBLL dan CE memberi ruang mati yang lebih kurang kerana kaedah CE adalah sepadan dengan CBLL. Ini bermakna kebarangkalian dalam CE adalah sama dengan ruang mati antara modal-modal. Akan tetapi, CE mengambil masa yang lebih panjang dibandingkan dengan GA kerana memerlukan lebih penjanaan rawak solusi pembolehubah iaitu  $10n^2$  di mana  $n$  mewakili bilangan modal.

## TABLE OF CONTENTS

ABSTRACT	ii
ABSTRAK	iv
TABLE OF CONTENTS	vi
TABLE OF FIGURE	x
TABLE	xiii
ACKNOWLEDGEMENTS	xiv
LIST OF SYMBOLS AND ABBREVIATIONS	xv
CHAPTER 1. INTRODUCTION	1
1.1 VLSI Design	1
1.2 Physical Design Cycle	3
1.2.1 Partitioning	3
1.2.2 Floorplanning and Placement	3
1.2.3 Routing	4
1.2.4 Compaction	5
1.2.5 Extraction and Verifiation	5
1.3 Automation in Floorplanning Optimization	8
1.4 Objectives and Goals	9
1.5 Research Outline	9
1.6 Importance of Research	10
1.7 Organization of Thesis	10

CHAPTER 2. LITERATURE REVIEW	11
2.1 Concepts of Floorplanning and Approaches to Problem	11
2.2 Floorplanning Representation	12
2.2.1 Slicing Floorplans	12
2.2.1.1 Slicing Tree	13
2.2.1.2 Normalized Polish Expression	14
2.2.2 Non-slicing floorplans	16
2.2.2.1 Corner Block List	17
2.2.2.2 O-Tree	22
2.2.2.3 B*-Tree	25
2.2.2.4 Bounded-Sliceline Grid	28
2.2.2.5 Sequence Pair	31
2.2.2.6 Corner Sequence	33
2.3 Optimization Algorithms	38
2.3.1 Simulated Annealing	39
2.3.2 Genetic Algorithm	40
2.3.3 Cross Entropy Method	42
CHAPTER 3. METHODOLOGY	43
3.1 Dot Model as Representation and Genetic Algorithm as Optimization Algorithm	43
3.1.1 Dot Model	44
3.1.1.1 DM to placement	44
3.1.2 Genetic Algorithm	50

3.1.2.1	Implementation of GA with DM	54
3.2	Floorplan Optimization using Corner Bottom Left List with Genetic Algorithm	65
3.2.1	Corner Bottom Left List	65
3.2.1.1	Preliminaries	66
3.2.1.2	From CBLL to Placement	67
3.2.2	Implementation of GA and CBLL	76
3.3	Floorplan Optimization using Corner Bottom Left List with modified Cross Entropy Method	78
3.3.1	Modified Cross Entropy Method	79
3.3.1.1	Cross Entropy Method	80
3.3.1.2	Random Generation of CBLL	82
3.3.1.3	Implementing Cross Entropy Algorithm	85
CHAPTER 4.	RESULTS, DATA ANALYSIS AND DISCUSSION	87
4.1	DMGA	87
4.1.1	Effects of mutation operators on floating point representation	87
4.1.2	Effects of crossover operators for floating point representation	89
4.1.3	Effects of mutation operators for ordered based sequence	91
4.1.4	Effects of crossover operators for ordered based numbers number	93
4.1.5	Optimal Results and Data Analysis	95
4.2	CBLL-GA	99
4.2.1	Effects of mutation operators on binary sequence	99

4.2.2	Effects of crossover operator for binary representation	101
4.2.3	Effects of mutation operators for ordered based sequence	102
4.2.4	Effects of crossover operators for ordered based numbers number	104
4.2.5	Optimal Results and Data Analysis	106
4.3	CBLL-CE	110
4.4	Discussion	114
CHAPTER 5.	CONCLUSION AND FUTURE WORK	119
5.1	Conclusion	119
5.2	Future Work	121
REFERENCES		122
APPENDIX A		128
APPENDIX B		133
APPENDIX C		136



## TABLE OF FIGURES

Figure 1.1 Physical design cycle .....	7
Figure 2.1 Slicing Floor plan .....	13
Figure 2.2 Slicing tree .....	13
Figure 2.3 Slicing structure, binary tree and the arithmetic expression of a floorplan .	15
Figure 2.4 Floorplan.....	18
Figure 2.5 Constraint Graphs .....	18
Figure 2.6 Deletion of Corner Block in a Floorplan Structure .....	19
Figure 2.7 Deletion of Corner Block in constraint graph.....	19
Figure 2.8 Insertion of Corner Block .....	20
Figure 2.9 Admissible Placement .....	22
Figure 2.10 Horizontal O-tree .....	22
Figure 2.11 When module is added on top, horizontal contour is searched from left to right and the top boundary of the new module is updated .....	24
Figure 2.14 BSG with dimension $p \times q$ , $BSG_{p \times q}$ .....	28
Figure 2.19 The corresponding placement.....	29
Figure 2.20(a) Placement, P (b) Contour R of P .....	35
Figure 2.21 (a) – (h) The process to build a CS from placement (i) CS representation .	36
Figure 2.22 (b) – (i) DSP packing scheme for CS in (a), where $CS =$ $\{(a,[s,t])(b,[a,t])(d,[a,b])(e[s,d])(c,[d,t])(f[e,c])(g[c,t])(h,[f,c])\}$ .....	37
Figure 3.1 DM solution string and shifting of the solution string.....	45
Figure 3.3 Placement from DM to floorplan.....	48
Figure 3.4 Crossover Operator .....	51
Figure 3.5 Mutation Operator .....	51
Figure 3.6 GA pseudocode.....	52
Figure 3.7 GA flowchart .....	53

Figure 3.8 Chromosome Model .....	54
Figure 3.9 Random Variables generation with repeated numbers .....	54
Figure 3.10 Random Variables from 1 to 4 is generated .....	55
Figure 3.11 Cyclic crossover .....	56
Figure 3.12 Uniform crossover .....	56
Figure 3.13 Partially mapped crossover .....	57
Figure 3.14 Order-based crossover .....	58
Figure 3.15 Single point crossover .....	58
Figure 3.16 Linear order crossover .....	59
Figure 3.17 Arithmetic Crossover equation .....	59
Figure 3.18 Heuristic crossover .....	60
Figure 3.19 Simple crossover .....	60
Figure 3.20 Inversion mutation .....	60
Figure 3.21 Swap mutation .....	61
Figure 3.22 adjacent swap mutation .....	61
Figure 3.23 three swap mutation .....	61
Figure 3.24 Shift mutation .....	62
Figure 3.25 Selection Probability .....	63
Figure 3.26 Placement of the first block with shape rectangle .....	68
Figure 3.27 Contour shape and their corners .....	69
Figure 3.28 Placing of block and updating the contour .....	70
Figure 3.29 Chromosome Model .....	76
Figure 3.30 Random Binary Variables .....	76
Figure 3.31 Random Binary Variables .....	76
Figure 4.1 Study on Number of Mutation Operators for floating point representation ..	88
Figure 4.2 Study on the Number of Crossover for Floating Point Representation .....	90

Figure 4.3 Study on the Frequency of Mutation for ordered based number .....	92
Figure 4.4 Study on the Frequency of Crossover for ordered based number .....	94
Figure 4.5 Graph showing the optimal results for DMGA .....	96
Figure 4.6 Placement for Apte .....	97
Figure 4.7 Placement for Xerox .....	97
Figure 4.8 Placement for hp .....	98
Figure 4.9 Placement for ami33 .....	98
Figure 4.10 Placement for ami49 .....	99
Figure 4.11 Study on Number of Mutation Operators for floating point representation .....	100
Figure 4.12 Study on the Number of Crossover for Floating Point Representation .....	102
Figure 4.13 Study on the Frequency of Mutation for ordered based number .....	103
Figure 4.14 Study on the Frequency of Crossover for ordered based number .....	105
Figure 4.15 Graph showing the optimal results for CBLL-GA .....	107
Figure 4.16 Placement for Apte .....	108
Figure 4.17 Placement for Xerox .....	109
Figure 4.18 Placement for hp .....	109
Figure 4.19 Placement for ami33 .....	110
Figure 4.20 Placement for ami49 .....	110
Figure 4.21 Graph showing the optimal results for CBLL and CE .....	111
Figure 4.22 Placement for Apte .....	112
Figure 4.23 Placement for Xerox .....	112
Figure 4.24 Placement for hp .....	113
Figure 4.25 Placement for ami33 .....	113
Figure 4.26 Placement for ami49 .....	114

## TABLE

Table 1: Packing and Placement .....	72
Table 2: Study on the Frequency of Mutation Operators.....	87
Table 3: Study on the Frequency of Crossover.....	90
Table 4: Study on the Frequency of Mutation .....	92
Table 5: Study on the Frequency of Crossover.....	94
Table 6: Optimal Results for DMGA.....	95
Table 7: Study on the Frequency of Mutation Operators.....	100
Table 8: Study on the Frequency of Crossover.....	101
Table 9: Study on the Frequency of Mutation .....	103
Table 10: Study on the Frequency of Crossover.....	105
Table 11: Optimal Results for CBLL-GA.....	107
Table 12: Optimal Results for CBLL and CE.....	111
Table 13: MCNC Benchmark Comparison.....	117

## **ACKNOWLEDGEMENTS**

I would like to express my gratitude and acknowledgement to Dr. Jeevan Kanesan, my supervisor, for his guidance and supervision in helping and guiding me in this project. With his advice, support and guidance, I am able to complete my research. I would like to thank Prof Velappa Ganapathy for his kind help in guiding me in writing papers for this research. I also want to thank Dr Harikrishnan Ramiah for his help when I needed advices.

I would also like to thank my mother, Tay Lee Lian and my father, Teoh Kim Yew, for giving me endless support throughout my studies in this university. Besides that, I would like to thank to my friends, Yeo Hock Chai, Hoo Chyi Shiang and Chong Wei Keat for helping and guiding me whenever I need them.

## **LIST OF SYMBOLS AND ABBREVIATIONS**

BSG	Bounded-Sliceline Grid
CBLL	Corner Bottom Left List
CE	Cross Entropy
CS	Corner Sequence
DIP	Dual In-Line Package
DFS	Depth-First Search
DM	Dot Model
GA	Genetic Algorithm
SP	Sequence Pair
VLSI	Very Large Scale Integration

## CHAPTER 1. INTRODUCTION

### 1.1 VLSI Design

Since 1960's, ICs were simple and consist of a few gates of flip-flop. Some of the ICs only perform logic function and use a single transistor with a resistor. Today's ICs expanded from a few transistors in a single chip to over more than 20 million transistors in a single chip and can run at the speed of GHz frequency. Besides that, MEM chips are also built to use for millions of electrical and mechanical devices. These chips bring a new era where exotic applications become reality such as tele-presence, augmented reality and implantable and wearable computer possible. This also gives a cost effective communication system to the whole wide world. (Sherwani, Naveed A., 2002)

Initially, the task of laying the gates and interconnects were done manually by drawing on graph papers and using layout editors. As the semiconductor fabrication processes improved, the number of transistors in a single chip increases and hence automation is needed to solve this addressing problem of increasing in transistor scales. Improvement in the computer speed enable to facilitate automation and it can be used for the next generation of computer chips to replace the current ones. (Alpert, Mehta, & Sapatnekar, 2009)

Initially, interconnect delays were not a factor and hence physical design is a simple process. The designer can place the blocks by floorplanning and then followed by placement to handle the rest of the logic. If the timing constraint of the design is met before placement, the timing constraints also will be met after placement. The increase in number of transistors causes more complications for designers as floorplanning becomes more complicated. Hence, algorithms and innovations are needed to aid in automated floorplanning. Floorplanning enables the designers to plan the input and outputs of the chip and also the global interconnect which is restricted to a given area

with the number of blocks in the circuit. This process needs to be done in a short time frame to determine the area for the chip that needs to be implemented physically. The process of determining the floorplan is important and hence automation for floorplanning optimization is introduced. (Alpert, Mehta, & Sapatnekar, 2009)

VLSI Physical Design Automation involves researching, developing and also producing algorithms and data structures to aid the physical design process. The main objective of this field is to arrange the devices in the circuit at an optimal arrangement on a plane. This is to achieve optimal interconnections between devices and obtain the best performance and the best functionality. Space on wafer is expensive, hence algorithm is to be developed to minimize space to reduce cost and to increase yield.

The arrangement of devices is important when determining the chip performance. Hence, algorithms for physical layout need to abide the rules required by the fabrication process. Fabrication rules are important to tolerate fabrication process. Algorithm must be efficient and able to handle very large designs. The efficiency of algorithm allows designers to save time and also enable designers to make iterative improvements to the layouts. The VLSI physical design process uses simple geometric objects to represent chips such as rectangle blocks. The physical design algorithm is similar to graph algorithms and hence combinational optimization algorithms can be used. Therefore, physical design automation can be studied from graph theory and also combinational algorithms that manipulate the geometric objects whether in two or three dimensions. However, geometric point of view will ignore the electrical aspect and design rules for physical design problems. Then, constraints must be implemented to suit the physical design problems.

Polygons and lines have inter-related electrical properties in VLSI circuit. This shows the complicated behaviour of VLSI design which also need to depend on various



variables which are required for IC design. It is necessary to keep both electrical aspects in taking geometric objects during the development of algorithms for VLSI physical design automation. (Sherwani, Naveed A., 2002)

## **1.2 Physical Design Cycle**

Input for physical design cycle is a circuit diagram and the output is the layout of a circuit. This cycle consists of a few stages which are partitioning, floorplanning, placement, routing and compaction. Figure 1.3 shows the stages of a physical design cycle followed by the explanations. Below are the explanations on the physical design cycle.

### **1.2.1 Partitioning**

Usually chips have many transistors. Because of the limitations in memory and computational power, it is difficult to plan the layout of the entire chip. Hence, chip is partitioned into sub-circuits. These sub-partitions are known as blocks. Partitioning process considers many factors such as the blocks size, the number of blocks and the number of interconnects between the blocks. Partitioning gives a set of blocks and interconnects between the blocks. Figure 1.1 (a) shows a circuit which has been partitioned into three blocks. Usually in large circuits, the process of partitioning is hierarchical and the topmost level of a chip may have 5 to 25 blocks.

### **1.2.2 Floorplanning and Placement**

In this step, good layout is selected for each block as well as the entire chip. The block area can be estimated after partitioning. Besides that, interconnect area within the block must also be taken into account. Rectangular shape that is determined by aspect ratio may vary within a pre-specified range. Generally, blocks usually have rectilinear shapes. Floorplanning is important to set up the ground work for a good layout. This step is computationally difficult and usually is done by design engineer rather than a

CAD tool. Usually human is better in observing the entire floorplan and analyse the information than a CAD tool. Sometimes manual floorplanning is needed for major component of an IC as the chip needs to be placed according to the signal flow of the chip. Moreover, some components need to be located at a specific location on the chip. Placement is when blocks are positioned on the chip. The aim of placement is to obtain the minimum area of the arrangement of blocks and also the complete interconnects between the blocks and meeting the constraints of the performance. Placement should be routable and meet their timing goals. There are two phases in placement which create initial placements and evaluate them and execute iterative improvements to obtain the minimum area and best performance according to the design specifications. Figure 1.1 (b) shows the placement of three blocks. Some spaces were left intentionally for interconnect between the blocks. The placement quality is analysed only after routing. Some placement may give unroutable design. Hence, another iteration of placement is needed. An estimation of a routing space is needed to limit the number of placement iterations. A good placement algorithm is important to obtain good routing and circuit performance. Little can be done to routing and the circuit performance once the position of the blocks is fixed.

### **1.2.3 Routing**

Routing is to do interconnect between blocks according to netlist. Firstly, the spaces which are not occupied by blocks are partitioned into rectangular regions which are called channels and switchboxes. The spaces between blocks and on top of the blocks are also included. The objective of routing is to connect all the blocks in the shortest wire length and only uses channels and switch boxes. There are two phases which are Global Routing and Detailed Routing. Global routing connects between proper blocks of the circuit disregarding the exact geometric details of each wire and pin. Global router will find a list of channels and switchboxes for every wire and uses it as a

passageway for the wire. Global routing specifies the routing spaces when a wire is routed. After global routing, detailed routing is done to complete the point-to-point connections between pins on the blocks. Global routing will be converted into exact routing according to the geometric information such as the location and space of wires and their layer of assignments. Detailed routing involves channel routing and also switches routing. Routing problems are computationally difficult. Hence, many researches have been done to solve routing problems which used heuristic algorithms. Several benchmarks have been standardized to evaluate the experiments using the algorithms. Sometimes, complete routing cannot be guaranteed. Hence, rip-up and re-route were sometimes used to remove some connections and reroutes them in a different order. Figure 1.1(c) shows the routing phase of all the interconnections between three blocks that have been implemented.

#### **1.2.4 Compaction**

Compaction is the task of compressing the layout in all directions to reduce the total area. Making the chip smaller will reduce wire lengths and reduce delay in signal between the circuits. Besides that, smaller area can produce more chips in a wafer and reduce manufacturing cost. However, computing compaction uses a lot of time and hence is only used for large volume applications such as microprocessors. It is important to ensure that compaction does not violate any design or fabrication rules. Figure 1.1 (d) shows a compacted layout.

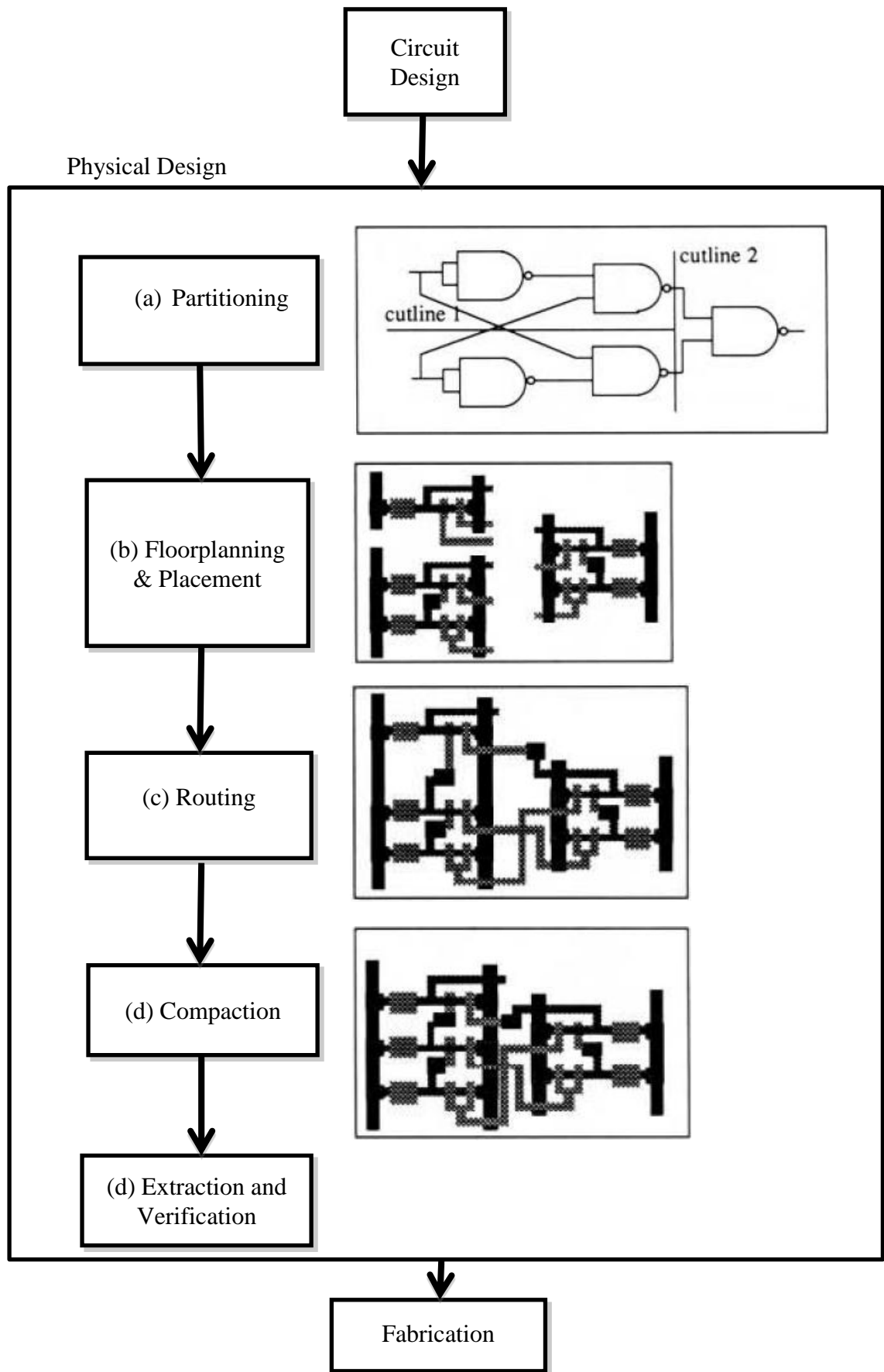
#### **1.2.5 Extraction and Verification**

It is important to have Design Rule Checking (DRC) to verify that all the geometric patterns follow the design rules for fabrication process. One of the rules that needed to be followed is the wire separation rule. Fabrication process requires a minimum specific separation between two adjacent wires. Hence, DRC needs to check that all the wires in the chip follow this rule. There are many other design rules and some of the rules are

difficult to check. After complete checking the design rules, the functionality of the layout needs to be verified by Circuit Extraction. This generates the circuit representation from the layout and is a reverse engineering process. The generated circuit is compared with the circuit description to check its correctness. This is known as Layout Versus Schematics (LVS) verification. The geometric information is extracted to compute Resistance and Capacitance. Hence, calculation for timing of each component and interconnect can be obtained and is known as Performance Verification. The information extracted is used to check the reliability aspects of the layout which is known as Reliability Verification. This is to ensure that the layout will not fail because of electro-migration, self-heat and other effects.

Same as VLSI design, physical design is also iterative and needs many steps and repeat many times to obtain better layout. Besides this, the quality of the solution is obtained from earlier steps. Early step such as placement will affect the routing. Hence, partitioning, floorplanning and placement are important to determine the chip area and performance compared to routing and compaction. This is because placements may give layout that cannot be routed. Hence, the chip needs to be repartitioned before routing again. Design cycle needs to be repeated several times in order to achieve design objectives.

The complexity of the design cycle depends heavily on the design constraints and also the design style used. This thesis is focused on floorplanning optimization where heuristic methods for floorplanning optimization will be analysed in order achieve the minimum floorplan area for physical design. (Sherwani, Naveed A., 2002)



### **1.3 Automation in Floorplanning Optimization**

Block arrangement is done in two phases which are floorplanning phase and the placement phase. Floorplanning is planning and sizing the blocks and interconnect whereas placement assigns a specific location of the blocks. For placement phase, it is important that the blocks are positioned on the surface of the layout so that no two blocks overlapping one another and there must be enough space for the interconnection. The blocks must be arranged so that the minimal total area of the layout is obtained.

The input for floorplanning phase is the set of blocks, area of each block, the shapes of blocks and the number of terminals for each block and the netlist. If the dimensions of the blocks are known, they are called fixed blocks and if the dimensions are not fixed, they are called flexible blocks. Floorplanning is usually generalized as the placement problem as the floorplanning involves flexible blocks whereas placement involves only fixed blocks.

There are a few factors needed to be considered during placements which are the shape of the blocks, the wire routing, the performance of the circuit for that floorplanning and placement, the packaging considerations and also the pre-placed blocks. (Sherwani, Naveed A., 2002)

Algorithms for VLSI floorplanning optimization are usually divided into 2 sections, the representation of the model and also the algorithm for optimization. There are various types of representations which can be used to represent floorplanning such as the floorplan tree and also the graph representations of floorplans. There are various algorithms such as the constructive method, iterative method and knowledge based method that can be used in order to optimize the floorplan through the use of the floorplanning representations. (Sait & Youssef, 1999)

## **1.4 Objectives and Goals**

The main objective of this research is on floorplan optimization. This research consists of floorplanning optimization using computational methods which consist of heuristic optimization algorithms and modelling floorplan representations. The objectives of this research are:

- 1) To analyse the type of methods that can be used to represent the floorplan.
- 2) To analyse the type of optimization method that can be used such as simulated annealing, genetic algorithm etc.
- 3) To introduce a representation for the floorplan to aid optimization.
- 4) To introduce a new algorithm that can be used to optimize the floorplan.
- 5) To obtain the minimum area of a given floorplan

## **1.5 Research Outline**

The outlines of this research consist of the following:

- 1) Obtain the different types of benchmarks that were used in previous research such as APTE, HP, AMI33 and AMI49.
- 2) Analyse methods of representing the floorplan such as sequence pair, graph theory and tree representation.
- 3) Analyse the different heuristic methods that are used to optimize the floorplan such as simulated annealing, genetic algorithm and particle swarm optimization.
- 4) Modelling the floorplan using sequence pair, corner sequence and also left bottom corner.
- 5) Optimizing the floorplan using genetic algorithm and cross entropy method by referring to the model of the floorplan.

## **1.6 Importance of Research**

Optimal automated floorplanning optimization is important to improve the layout in a floorplan so that the chips will be at their best performances and also reduces the costs of manufacturing and man power. This is because less design engineers will be needed as automated algorithm is able to aid IC design.

## **1.7 Organization of Thesis**

This thesis is divided into six main chapters. The first chapter gives a brief introduction on VLSI design and also the importance of automation for VLSI design. The second chapter discusses on floorplanning in a more detailed way and also reviews about previous work done in order to solve the floorplanning optimization problem. The third chapter of this thesis discusses on the approaches taken for this project which are Genetic Algorithm (GA), Cross Entropy Method (CE), Dot Model (DM) and Corner Bottom Left List (CBLL). The fourth chapter discusses on the results obtained in this work compared with results obtained by previous researches. The fifth chapter are the results and discussions of this research work. The final chapter concludes about this project and also the future work that can be carried out in order to improve this project.



## **CHAPTER 2. LITERATURE REVIEW**

### **2.1 Concepts of Floorplanning and Approaches to Problem**

Floorplanning is important in VLSI physical design automation. Floorplanning arranges a set of rectangular modules of different sizes and find the placement of these modules in a way where no module overlaps each other and is arranged in a minimum area and minimum wire length. The abstract formulation involves rectangular blocks with arbitrary dimensions.

The main objective of floorplanning is to obtain the minimum space of layout in order to save cost and also helps to reduce wire routing and reduce circuit resistance. Reduction in resistance will lead to less heat generated. This can further improve the performance of the VLSI chip. In order to optimize floorplanning, coding the floorplan is important to use optimization algorithm. There are several floorplanning methods which were used as follows:

- 1) Constraint based methods
- 2) Integer programming based methods
- 3) Rectangular dualization based methods
- 4) Hierarchical tree based methods
- 5) Simulated Evolution algorithms
- 6) Timing Driven Floorplanning Algorithms

Besides the methods mentioned above, there are many other methods either simple or more complicated which can be used for floorplan optimization. In this project, a stochastic method is used for optimization. (Sherwani, Naveed A., 2002) To optimize a floorplan, it is essential to have a good representation code. This is important so that the optimization algorithm can use the string based representation in order to

obtain optimum arrangement of the layout. This shows that floorplanning consists of 2 main sections which are floorplan modelling and floorplan optimization.

In the recent years, many researches have been carried out and developed for floorplanning optimization. There are various floorplan representations that have been developed which are non-topological, room-based and also block-based. Generally, floorplan can be categorized into two main types which are the slicing and non-slicing floorplan. In this chapter, previous methods of floorplanning representations and optimization methods will be discussed.

## **2.2 Floorplanning Representation**

In this section, the various types of floorplan representations will be discussed in detail. Slicing floorplan representations are represented by Polish expression and slicing tree. Non-slicing floorplan representations are corner block list (CBL), Sequence Pair (SP), Bounded Slice-line Grid (BSG), O-tree, B\*-tree, Transitive Closure Graph (TCG), Transitive Closure Graph with a Sequence (TCG-S) and Adjacent Constraint Graph (ACG).

### **2.2.1 Slicing Floorplans**

Slicing floorplan can be partitioned into at least 2 different blocks. Slicing floorplans are hierarchical floorplans of order 5. A floorplan is hierarchical of order 5 only if it can be recursively subdivided into rectangle either two parts by a horizontal or into 4 parts by a wheel.

### 2.2.1.1 Slicing Tree

A slicing tree is a binary tree that has  $n$  leaves and  $n-1$  nodes, where each of the nodes represents a vertical or horizontal cut line and each of the leaves represents a block. Slicing tree is also known as slicing floorplan tree. Figure 2.1 shows the slicing floorplan and Figure 2.2 shows the tree representation for the slicing floorplan of Figure 2.1. In the slicing tree, the internal nodes are labelled with either V or H which means vertical or horizontal cut. Every leaf is labelled by the module number. A slicing tree is skewed if only it has no node and the right child has the same label. Figure 2.2a shows a skewed slicing tree and Figure 2.2b shows non-skewed slicing tree. Slicing floorplan can be represented by more than one slicing trees. The order of horizontal and vertical cuts makes up the slicing tree. Skewed slicing tree is unique for a slicing floorplan.

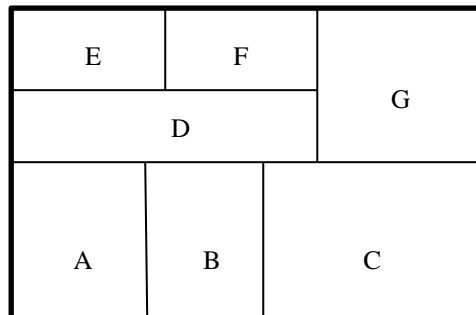


Figure 2.1 Slicing Floor plan

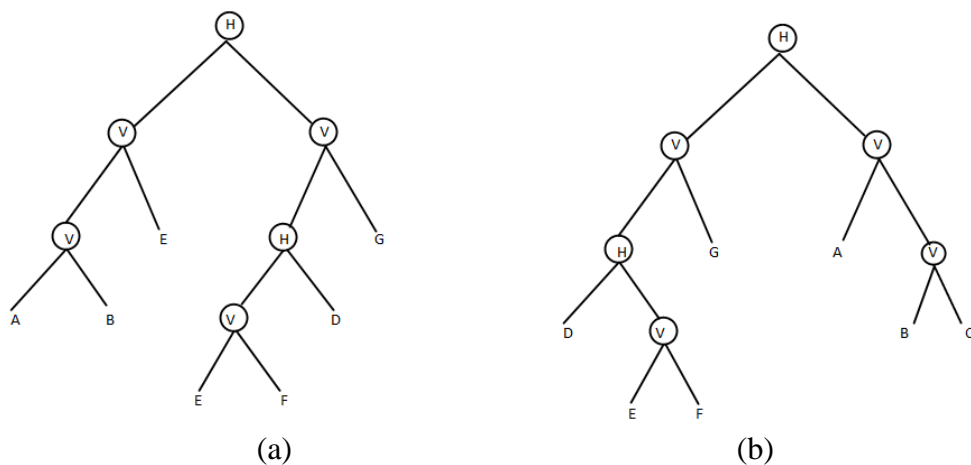


Figure 2.2 Slicing tree

Slicing tree is a top-down description of the type of cut which is horizontal or vertical of a slicing floorplan. Slicing tree does not give dimensional information. Hence, a slicing tree may represent more than a slicing floorplan. General slicing floorplan is known as hierarchical floorplan of order 5. An order 5 floorplan can be recursively subdividing 2 parts by a horizontal or vertical segment or into 4 parts by a wheel. Hence, Figure 2.1 is known as hierarchy of order 5. (Sait & Youssef, 1999)

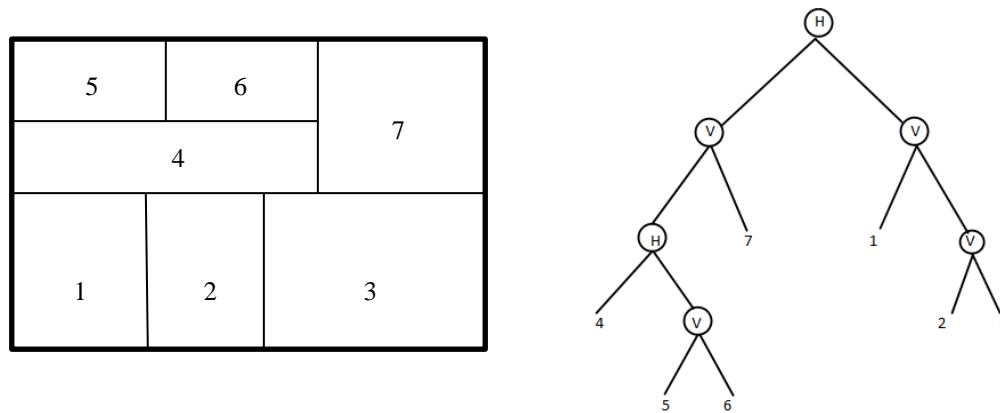
According to (Lai & Wong, 2001), slicing tree can be an effective tool for VLSI floorplan design where it can take full advantage of shape and orientation flexibility of circuit modules to find highly compact slicing floorplan. They mentioned that using slicing tree representation and compaction, all maximally compact placements of modules can be generated and it becomes a complete floorplan representation for all non-slicing floorplans.

#### **2.2.1.2 Normalized Polish Expression**

Normalized Polish Expression uses rectangle dissection where a rectangle is subdivided by horizontal and vertical line segments into more than one non-overlapping rectangle. This can be represented in slicing tree. The normalized Polish expression of length  $2n-1$  has a 1-1 corresponding with the slicing trees with  $n$  leaves. A polish expression also is a top down description. The symbols for H (horizontal) and V (vertical) are the operands for the slicing structures. Hence, if A and B are slicing structures, it can be represented with an “arithmetic expression” with  $AHB$  or  $AVB$  which are the representation for Polish expression. Figure 2.3 shows the slicing structure, binary tree and the arithmetic expression of a floorplan.

A skewed slicing tree is a slicing tree where no node and its right child has the same label. A Polish expression  $a_1, a_2, \dots, a_{2n-1}$  is normalized if and only if there is no consecutive  $H$  or  $V$  in the sequence. Polish expression corresponds to the slicing

structure and also slicing tree. This causes a slicing structure can be represented by more than two Polish expressions.



$((4H(5V6))V7)H(1V(2V3))$  - "Arithmetic Expression"

Figure 2.3 Slicing structure, binary tree and the arithmetic expression of a floorplan

Normalized polish expression has three properties which are

1. each block appears only once in a string
2. the balloting sequence property
3. no two consecutive operators are the same in a string

These properties of normalized polish expression will give a unique slicing floorplan for a normalized polish expression. Normalized polish expression enables us to perturb the slicing floorplan solution in order to obtain a new neighboring solution. Polish expression can be evaluated with a  $O(n \log n)$  bottom-up traversal with the corresponding slicing tree. Hence, both area and wirelength of the floorplan can be obtained. This is important for evaluation to reduce optimization time. (Wong & Liu, 1986)

For VLSI floorplanning, some placement constraints are specified in packing. One type of placement constraint is to pack some modules on one of the four sides: on the

left, on the right, at the bottom, or at the top of the final floorplan. These are called boundary constraints. Young, Wong, & Yang, 1999 enhanced a well-known slicing floorplan algorithm which is the Polish expression that represents the intermediate solutions in the simulated annealing process, so that constraints can be checked and fixed efficiently. (Young, Wong, & Yang, 1999)

Another paper (Lin, Chen, & Wang, 2002) proposed generalization of Polish expression where the representation can efficiently reuse some area that cannot be utilized if only have vertical and horizontal operators defined in Polish expression and hence Polish expression can represent non-slicing floorplans.

This paper (Chen, Lin, & Wang, 2003) addresses the problem of VLSI floorplanning with boundary constraints consideration. They use generalized Polish Expression which uses both Polish Expression and also the boundary constraints for non-slicing floorplan. Besides that, a fixing heuristic based on modular similarity is presented to effectively fit the generated infeasible floorplans during the process. Hence, this new Polish Expression is modified from Polish Expression so that non-slicing floorplan can be represented.

### **2.2.2 Non-slicing floorplans**

Non-slicing floorplan are floorplans that cannot be divided by horizontal or vertical cuts. Hence, the smallest nonslicing floorplan is a wheel. Non-slicing floorplans can give more optimum area as it does not need to be arranged in a way that it needs to be divided into smaller rectangular. Below are the non-slicing floorplans and their description.

### 2.2.2.1 Corner Block List

Corner block list is a topological representation for non-slicing floorplan. Corner block list takes linear time to construct the floorplan. It defines the floorplan independent of the block sizes and is able to optimize various size configurations of block. Corner block list time complexity to convert into floorplan is  $O(n)$ . Corner block list takes  $n(3 + \lceil \lg n \rceil)$  bits to describe where  $\lceil \lg n \rceil$  denotes the minimum integral number which is not less than  $\lg n$ . Besides that, corner block list represents the floorplan independent of the block sizes and hence this can optimize blocks with different widths and heights.

Corner block list is constructed based on the recursive corner block deletion. Every block deletion is kept according to the block name, corner block orientation and number of T-junctions uncovered. Upon completion of the deletion iterations, the data of these three items are concatenated in a reverse order. The sequence of the block names  $S$ , the list of orientations  $L$  and the list of T-junction  $T$  information is then obtained. At the  $n$ th deletion, only one block is left in the floorplan. Hence, the orientation and the number of T-junction can be ignored and were not included in the lists of  $L$  and  $T$ .

A constraint graph for a floorplan is represented as  $G = (V, E)$ , where the nodes in  $V$  are segments which slice the space and form rooms for the floorplan with nodes used for the placement boundaries and  $E$  are the edges of the room of placement blocks. There are two types of edges where one direction is from left node to right node and another direction is from bottom to top node. Source node of an edge is the outgoing edge and destination node of an edge is incoming edge.

There are two constraint graphs which are the horizontal constraint graph (HCG) and vertical constraint graph (VCG). For HCG, “W” represents west pole and “E”

represents east pole. These edges represent the direction related horizontally from left to right. For VCG, “S” represents south pole and “N” represents north pole. The edges represent the direction related vertically from bottom to top. Figure 2.4 shows the floorplan and Figure 2.5 shows the constraint graphs related to Figure 2.4. Edge that points to east or north pole is known as corner edge. A block that is at the corner edges of both HCG and VCG is corner block. Hence, only block “d” in Figure 2.4 is corner block. Orientation of corner block is defined according to the joint in left and bottom segment and T-junction. T-junction has 2 orientations which are rotated by 90 degrees and 180 degrees counterclockwise. The corner block is vertical oriented if T is rotated by 90° and is denoted by “0”. The corner block is horizontal oriented if is rotated by 180° and is denoted by “1”. Figure 2.4 shows that the corner block, d is vertical oriented and is denoted as “0”.

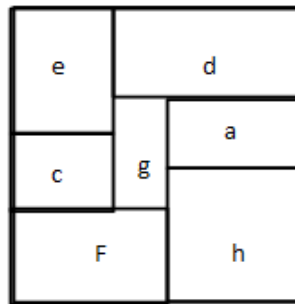


Figure 2.4 Floorplan

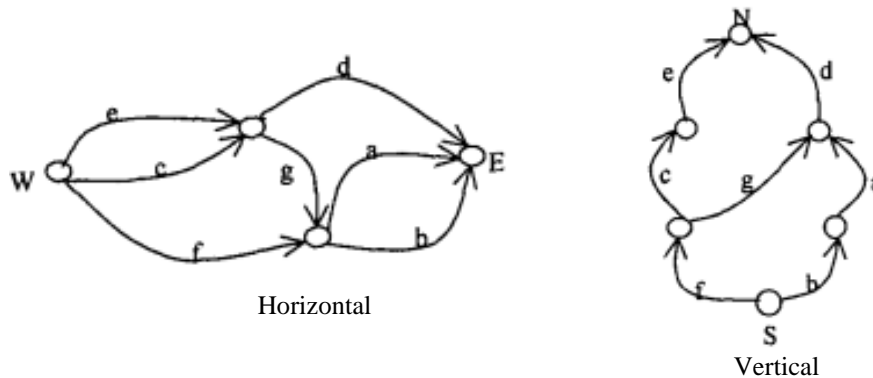


Figure 2.5 Constraint Graphs



Corner Block Deletion is deleting the right top corner block. For horizontal oriented corner block, when the corner block is deleted, the left segment is shifted to the right boundary of the chip and the attached T-junction is pulled along with the segment. For vertical oriented corner block, when the corner block is deleted, the bottom segment is shifted to the top boundary of the chip and the attached T-junction is pulled along with the segment. Figure 2.6 shows corner block deletion for horizontal oriented corner block. Hence the corresponding constraint graphs can perform deletion directly. The constraint graph will be modified as shown in Figure 2.7. This will modify the constraint graph and block “a” becomes the corner block.

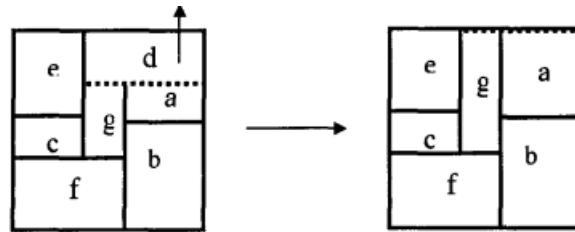


Figure 2.6 Deletion of Corner Block in a Floorplan Structure

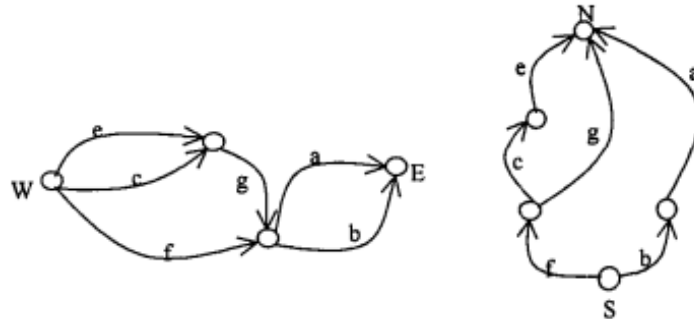


Figure 2.7 Deletion of Corner Block in constraint graph

Corner block insertion is inserting block into the right top corner. When the corner block to be inserted is vertical oriented, the horizontal segment at the top side of the chip is pushed down to cover the designated set of T-junctions and to obtain room to insert the corner block as shown in Figure 2.8. When the corner block to be inserted is horizontal oriented, the left vertical segment is pushed at the right side of the chip and then the corner block is inserted. The floorplan still remains mosaic after deletion and insertion.

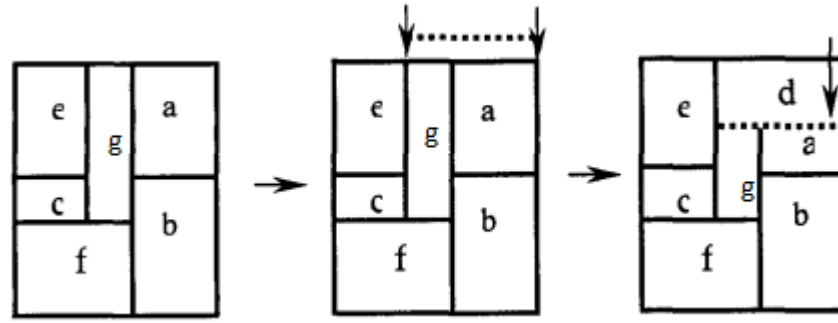


Figure 2.8 Insertion of Corner Block

Corner Block List has three tuples which is (S,L,T). For Figure 2.3 floorplan, the corner block, d is deleted. Since d is a vertical oriented block and there is one T-junction attached to it, the record obtained is (d,0,10). Hence, the block a,b,g,e,c and f are deleted successively. Hence, the concatenate record derive a corner block list of (S,L,T) where  $S = (fcegbad)$ ,  $L=(001100)$ , and  $T = (001010010)$ .

Below is the algorithm that transforms from floorplan to corner block list:

When corner block exist, repeat

1. Delete corner block
2. If not the last corner block, record the block name, orientation and T-subsequence.

Add the last block to the list and concatenate the records in a reverse order of the deletion sequence to construct the corner block list.

Below is the algorithm that transforms from corner block list to floorplan:

1. Initialize the floorplan with block  $S[1]$
2. For  $i=2$  to  $n$ , insert block  $S[i]$  with orientation  $L[i]$  and T-junctions from the corner block list. If the number is more than the T-junctions available, exist and report error,

A corner block list may not correspond to floorplan because of the constraints of list T. Hence, the number of erased T-junction cannot be more than the T-junctions available for insertion. (Hong, et al., 2000)

This paper (Dhamdhere, Zhou, & Wang , 2002) uses CBL representation for module placement with pre-placed modules. They use two methods. First method, only free modules included in the corner block list. The free module from the CBL is placed and check overlaps for the pre-placed modules and removes the overlaps by shifting the free module to the right or top. The second method uses all modules in the corner block list. If a new module inserted is found to be overlapping with pre-placed module, it is instead swapped with the pre-placed module in the CBL. If a new module inserted is a pre-placed module but not in the pre-placed location, the placement of the pre-placed module will be deferred and is swapped with the next free module in the CBL to place at the current location. These algorithms are combined with simulated annealing technique.

Another paper by (Hong, Dong, Huang, Cai, Cheng, & Gu, 2004) uses CBL to represent mosaic floorplans. In mosaic floorplan, each room has only one block assigned to it. Hence, a unique corner room is available on the top right corner of the chip. The corner block deletion and corner block insertion will be used to keep the floorplan mosaic. Recursive deletion process can convert the mosaic floorplan to a representation named CBL. This CBL uses linear time to construct the floorplan. Simulated annealing is used for optimization.

According to (Chen, Dong, Hong, Ma, & Cheng, 2006), CBL is a room-based floorplan representation. This paper identifies the topological relation between two blocks in CBL. Using the topological relation between the blocks, CBL is feasible under alignment constraints. Hence, the block placement can be done using CBL with alignment constraints.

### 2.2.2.2 O-Tree

O-tree is an ordered tree that represents non-slicing floorplans. O-tree model is used for admissible placement. An admissible placement is a placement where the modules can only be compacted in both x- and y- directions. This means that the module cannot shift left or down with other modules were being fixed. Figure 2.9 shows an admissible placement.

A tree has a finite set of  $T$  or more nodes. It has a specially designated node which is the root of the tree. The root has zero or more branches that are pointing from the root children. O-tree has two types which are horizontal O-tree and vertical O-tree. To construct O-tree, admissible placement is needed. A horizontal O-tree  $(T, \pi)$  is constructed as follows. Left boundary of the placement is represented as root and the x-coordinate is set as  $x_{\text{root}} = 0$  and width  $w_{\text{root}} = 0$ . The children are placed on the right side of their parent with zero separation distance in x direction. Figure 2.10 shows a horizontal O-tree for the floorplan structure of Figure 2.9. (Takahashi, Guo, Cheng, & Yoshimura, 2003)

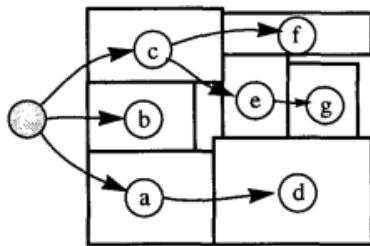


Figure 2.9 Admissible Placement

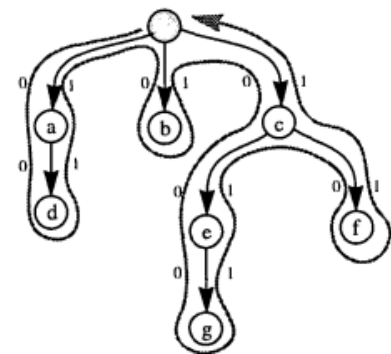


Figure 2.10 Horizontal O-tree

A vertical O-tree is constructed by having the bottom boundary of the placement as the root and the edge gives the direction relationship between the modules. An O-tree is represented by two-tuple  $(T, \pi)$ . In order to encode a rooted ordered tree with  $n$  nodes,

a  $2(n-1)$ -bit string,  $T$  branching structure needs to be identified and the permutation,  $\pi$  to label the  $n$  nodes. The bit string,  $T$  realizes the tree structure. “0” stands for traversal descends edge and “1” stands for traversal ascends edge in the tree. The permutation,  $\pi$  labels the sequence to traverse the tree in depth-first search order. The root of the tree is represented by the first element in the permutation,  $\pi$ . Figure 2.10 shows the encoding of 8-node rooted ordered tree. The root node has three subtrees which are a, b and c. The O-tree can be represented as (00110100011011,adbcegf).

The left boundary of the floorplan is set as the root of a horizontal O-tree and the coordinate is set  $(x_{\text{root}}, y_{\text{root}}) = (0, 0)$ . Node  $n_i$  is the parent of node  $n_j$ . Hence,  $x_j = x_i + w_i$ . For each block  $b_i$ , let  $L(i)$  be the set of block  $b_k$ 's on the left of  $b_i$  in permutation  $\pi$ , and interval  $(x_k, x_k + w_k)$  overlaps the interval  $(x_i, x_i + w_i)$  by a nonzero length. If  $L(i)$  is non-empty, we have

$$y_i = \begin{cases} \max_{k \in L(i)} \{y_k + h_k\}, & L(i) \neq \emptyset \\ 0, & \text{otherwise} \end{cases}$$

Horizontal O-tree can give placement by visiting the tree in DFS order. This is shown in Figure 2.10. Y-coordinate can be computed from horizontal O-tree by using contour structure to reduce the run time to find y-coordinate of a block. Without contour structure, the run time is linear to the number of blocks without contour structure. Contour structure can find the y-coordinate in a constant time. Contour structure is double linked list of blocks that describe the contour line in the current compact direction. Figure 2.11 shows the contour structure is updated when new block is placed into the floorplan and the y-coordinate of the block is determined. (Guo, Cheng, & Yoshimura, 1999)

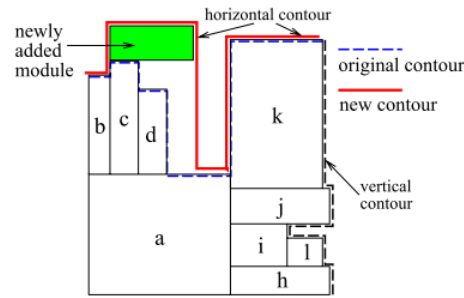


Figure 2.11 When module is added on top, horizontal contour is searched from left to right and the top boundary of the new module is updated

According to Sitzmann & Stuckey, 2000, constraint search trees is used to search trees where the operations are defined in term of constraints. Hence, the fundamental operations of search trees and the immediate points to new possibilities for search trees are made clear. A height-balanced constraint search tree which builds a height-balanced index structure which is O-tree. N object in an O-tree is represented by constraints of the form  $ax_i + bx_i \leq d$  where  $\{a,b\} \subseteq \{-1,0,1\}$  and  $x_1, \dots, x_n$  are the dimension of the spatial data. Hence, the basic operations to build and search the height-balanced constraint search tree and constraint joins are defined by (Sitzmann & Stuckey, 2000). Hence, accurate information in the O-tree nodes can be obtained.

A modified O-Tree based packing algorithm is also proposed. (Yan, Li, Yang, & Yu, 2004) This can reduce the time cost of turning an O-Tree into an admissible O-Tree and decoding it into a placement. This is done by introducing a vertical contour into the algorithm by replacing the O-Tree orthogonal constraint graph and constraint graph to O-Tree procedures to a single O-Tree to floorplan. This algorithm applied to various problems such as placement with boundary constraint and rectilinear blocks. Genetic Algorithm is used to solve the problem and an additional block is inserted as internal node without increasing the time complexity when perturbation of O-Tree.

There is a paper (Ninomiya, Numayama, & Asai, 2006) which also uses O-tree for floorplan optimization. This work shows a two-staged Tabu search for non-slicing

floorplan problem using O-tree. This combines the simulated annealing into the two-staged Tabu search and a hybrid algorithm which is the O-tree representation. This paper combines both the optimization algorithms with O-tree for floorplan optimization.

### 2.2.2.3 B\*-Tree

B\*-trees are based on ordered binary trees and the admissible placement. Inheriting the properties of ordered binary trees, B\*-trees can be implemented easily and can perform respective primitive tree operation search, insertion and deletion in only constant, and linear time.

The correspondence between an admissible placement and induced B\*-tree is direct and only takes linear time. The evaluation for B\*-tree and placement can be done directly and incrementally. This reduces the search spaces and avoids redundant solutions. (Chang, Chang, Wu, & Wu, 2000)

An admissible placement,  $P$  can be represented by a unique (horizontal) B\*-Tree  $T$ . Figure 2.12 shows the B\*-tree of a floorplan of Figure 2.13. A B\*-tree is an ordered binary tree where the root is directly related to the module of the bottom-left corner. B\*-tree is constructed in a recursive way.

Initially, the left subtree is constructed recursively to the right subtree from the root. Let  $R_i$  denote the set of modules located on the right-hand side and adjacent to  $b_i$ . The left child of the node  $n_i$  corresponds to the lowest module in  $R_i$  that is unvisited. The right child of the node  $n_i$  represents the lowest module located above and with its  $x$ -coordinate equal to that of  $b_i$ . The  $y$ -coordinate is less than half the top boundary of the module on the left hand side and adjacent to  $b_i$ . This display the corresponding relationship between the admissible placement and B\*-tree.

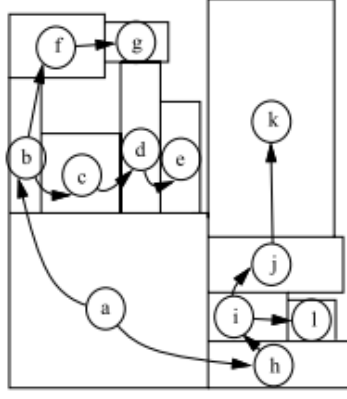


Figure 2.12 An admissible placement

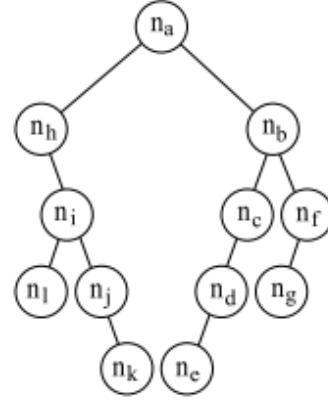


Figure 2.13 B\*- tree

Referring to Figure 2.12 and Figure 2.13, module  $a$  is the root because it is on the bottom-left corner. The left subtree of  $n_a$  is constructed recursively making  $n_b$  the left child of  $n_a$ . The procedure is repeated for the right subtree of  $n_a$  once the left subtree of  $n_a$  is completed. This construction only needs linear time.

A B\*-tree can be computed into the  $x$ -coordinate and  $y$ -coordinate for each module according to the node in the tree. The root  $(x_{\text{root}}, y_{\text{root}}) = (0, 0)$  for the  $x$ - and  $y$ -coordinates of the module because the root of  $T$  is the bottom-left module.

B\*-tree keeps the geometric relationship between two modules. If node  $n_j$  is the left child of node  $n_i$ , module  $b_j$  will be located on the right-hand side and adjacent to module  $b_i$  in the admissible placement; i.e.,  $x_j = x_i + w_i$ . Whereas, if node  $n_j$  is the right child of  $n_i$ , module  $b_j$  will be located above and adjacent to module  $b_i$  with the  $x$ -coordinate of  $b_j$  equal to that of  $b_i$ ; i.e.,  $x_j = x_i$ .

The construction is recursively performed in depth-first search order. A B\*-tree can determine the  $x$ -coordinates of all the modules by traversing the tree completely once. The contour data structure can also be adopted to compute the  $y$ -coordinate of a B\*-tree as mentioned in the O-tree. (Chang, Chang, Wu, & Wu, 2000)



A paper by Jiang, Lai, & Wang studied the problem of module placement with pre-placed modules and it extends the B\*-tree representation to solve problem. This paper suggests that each pre-placed modules to be placed at a pre-specified location all the time and uses the B\*-tree representation to generate the locations for the remaining modules. A repositioning techniques is added in order to eliminate any overlapping between modules. This paper added simulated annealing to optimize the placement. (Jiang, Lai, & Wang, 2001)

Wu & Chang used B\*-tree representation for placement with alignment and performance constraint. The aim is to align circuit blocks one by one and constrain the blocks within a certain bounding box. The feasibility conditions is explored with the alignment and performance constraints and then an algorithm is used to guarantee a feasible placement with alignment constraints and generate a good placement with performances constraints during each operation. (Wu & Chang, 2004)

B\*-tree representation can also incorporate with particle swarm optimization (PSO) for floorplanning. The B\*-tree floorplan structure is used to generate an initial stage with overlap free for placement and PSO to find potential optimal placement solution. This method can avoid solution from falling into the local minimal and able to explore good solutions for floorplan optimization. (Sun, Hsieh, Wang, & Lin, 2006)

Mao, Xu and Ma used hybrid algorithm which incorporate B\*-tree representation for floorplanning. This is to improve the area optimization of the floorplan. This uses simulated annealing embedding a tabu search for the floorplan. The purpose of this hybrid algorithm is to improve area utilization and obtain the optimal results in a short run time. (Mao, Xu, & Ma, 2009)

#### 2.2.2.4 Bounded-Sliceline Grid

A bounded-sliceline grid (BSG) dissects a plane into rooms what are associated with binary relations of “right to” and “above” where any two rooms are in this unique relation. This can be obtained through an assignment of modules on the BSG followed by the physical realization of the BSG-PACK.

BSG is a metagrid where it means that it does not contain physical dimension but is a topological grid of plane that has orthogonal unit lines which are called BSG-units. BSG dissects plane into rectangular zone which are known as rooms. Hence, BSG structure has rooms, horizontal unit segments and vertical unit segments. Figure 2.15 shows the a BSG of dimension  $p \times q$ ,  $BSG_{p \times q}$ . To use BSG structure to represent a floorplan,  $p \times q$  must be equal or larger than the number of modules. A rectangular space which is surrounded by an adjacent pair of vertical and horizontal units will form a room. The vertical unit segment gives the vertical relations and the horizontal unit segment gives the horizontal relations. The placement for modules,  $m$  is arranged as the room assignment of modules,  $m$  by placing the modules,  $m$  into different rooms. Given a set of modules,  $M$  where  $|M| = n$ . Making an assumption that  $p \times q \geq n$ , an assignment of  $M$  is a one-to-one mapping of modules into the rooms of  $BSG_{p \times q}$ . A room that does not have module assigned is called an empty room.

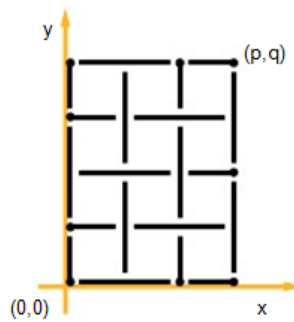


Figure 2.14 BSG with dimension  $p \times q$ ,  $BSG_{p \times q}$



The horizontal unit adjacency graph is represented by  $G_h(V_h, E_h)$ . For each vertex  $u \in V_h$ ,  $l_h(u)$  is denoted as the length of the longest path from the source  $s_h$  to  $u$ . For the vertical unit adjacency graph,  $G_v(V_v, E_v)$ , each vertex  $u \in V_v$ ,  $l_v(u)$  denotes the length of the longest path from the source  $s_v$  to  $u$ . The longest path,  $G$  is used to determine the positions of the modules. This longest path algorithm works in linear time with the number of edges when the input  $G$  is a directed acyclic graph. The total number of edges of the unit adjacency graphs are between  $2(pq + p + q)$  and  $2(pq + p + q) - 4$ . The time complexity of the longest path length,  $G$  where  $G$  is  $G_h(V_h, E_h)$  or  $G_v(V_v, E_v)$  is  $O(pq)$ .

According to the BSG to placement theorem, the following procedure is done according to the procedure of BSG-PACK as follows when given an assignment of  $M$  to  $BSG_{pq}$ . Let the module,  $m$  is assigned to a room where the boundary of the left vertical unit is  $V_m$  and the bottom horizontal unit is  $H_m$ . Then, module,  $m$  is placed at the left bottom which is at  $(l_v(u_{V_m}), l_h(u_{H_m}))$  where  $u_{V_m}$  and  $u_{H_m}$  are the vertices that corresponding to the units  $V_m$  and  $H_m$  where  $V_m$  is the vertical unit and  $H_m$  is the horizontal unit adjacency graph.

It is important to ensure that no two modules are overlapped by noticing the right to and above relation in the output of the procedure. From there, we can obtain the area which is the minimum bounding box of the chip by using the BSG-PACK assignment which is  $(l_v(t_v) \times l_h(t_h))$ . The procedure of BSG-PACK gives the physical dimension to the meta-grid. The output of BSG-PACK can be further compacted to reduce area if the modules are allowed to penetrate the peripheral BSG-units. BSG can be perturbed by choosing two different rooms and interchange the contents in them to generate a new BSG assignment.

### 2.2.2.5 Sequence Pair

Sequence Pair encodes non-slicing floorplans. When  $n$  modules of a non-slicing floorplan are given, the sequence pair, which is a pair of module name sequences will provide the information regarding the position of the modules relatively whether it is above, to the right of and to the left of a given module. The correspondence between sequence pair and placement is 1-to-1 and hence a sequence pair gives a unique floorplan.

In order to obtain a floorplan from the sequence pair, first draw an 'up-right step-line' for each module. Vertical lines and horizontal lines were drawn from the upper right corner of a module until the upper right corner of the floorplan. The down-left step-lines are also drawn in a similar way. A pair of up-right step-line and down-left step-line for a given module forms the 'positive step-line' and a pair of left-up and right-down step-line for a given module forms the 'negative step-line.' Positive step-lines of the modules and negative step-lines of the modules do not cross each other. The order among the positive step-lines from the left to right forms the first sequence in a sequence pair and the order among the negative step-lines from the bottom to top gives the second sequence in the sequence pair.

In order to obtain sequence pair to placement, the following steps are taken. Given a module  $x$  in a sequence pair  $SP(S_1, S_2)$ , the list of modules that appear before  $x$  in both  $S_1$  and  $S_2$  is obtained. These modules are located on the left of  $x$  in the floorplan. The set of modules that appear after  $x$  in both  $S_1$  and  $S_2$  are located to the right of  $x$  in the floorplan. The set of modules that appear after  $x$  in  $S_1$  and before  $x$  in  $S_2$  are located above  $x$  in the floorplan. Then a directed graph, Horizontal Constraint Graph (HCG) is built based on the 'right-of' and 'left-of' relation, where a directed edge  $e(a,b)$  shows that module  $a$  is to the left of  $b$ . Then, source node is added and is connected to all the

nodes in HCG. A sink node is also added to HCG and is connected to all nodes to this sink. A longest path length from source to each node in HCG denotes the x coordinate of the module in the floorplan. The longest source-sink path length gives the width of the floorplan. To construct a Vertical Constraint Graph (VCG) also is the same as HCG but it uses the 'above' and 'below' relation and computes the y coordinates of the modules and gives the height of the floorplan. (Murata, Fujiyoshi, Nakatake, & Kajitani, 1995)

According to Tang, Tian and Wong, a fast evaluation of a large number of sequence pairs is required in order to evaluate each generated sequence pair into corresponding block placement. Hence, they suggest that a new approach to evaluate a sequence pair based on computing longest common subsequence in a pair of weighted sequences. This is to improve the efficiency of sequence pair to  $O(n^2)$  algorithm. The aim is to reduce the runtime for floorplan optimization. (Tang, Tian, & Wong, 2001)

According to Kodama and Fujiyoshi, sequence-pair can be used as a representation of block placement to determine the densest possible placement of rectangular modules in VLSI layout design. They suggested that a method of obtaining packing via the Q-sequence (Representation of rectangular dissection) in  $O(n+k)$  time from a given sequence pair of  $n$  rectangles with  $k$  subsequences called adjacent crosses, given the position of adjacent crosses and the insertion order of dummy modules in adjacent courses. This method keeps the  $k$  not more than  $n-3$ . (Kodama & Fujiyoshi, An Efficient Decoding Method of Sequence Pair, 2002)

Another method of sequence pair which is proposed is selected sequence-pair (SSP), a sequence pair with limited number of subsequences called adjacent crosses. This is a modification made from sequence pair. This method has smallest packing based on a given SSP can be obtained in  $O(n)$  time, where  $n$  is the number of rectangles

and can represent arbitrary packing. The total representation number of SSP of size  $n$  is not more than the rectangular dissection of the same size. SSP is incorporated with an algorithm in order to enumerate all the adjacent crosses on a sequence pair in linear time. Besides that, they convert a sequence pair without adjacent crosses to an equivalent Q-sequence, representation of rectangular dissection as mentioned in the previous method. (Kodama & Fujiyoshi, Selected Sequence-Pair: An Efficient Decodable Packing Representation in Linear Time using Sequence Pair, 2003)

There is a suggestion where floorplanning optimization using sequence pair as representation is incorporated with genetic algorithm. The sequence-pair is a data structure with applications in packing-based VLSI module placement. This paper uses genetic algorithm for rectangle packing. There is an extent to handle symmetry constraints which is a requirement for analog circuits. There are genetic operators which were developed to accommodate the specific properties of the sequence pair. (Drakidis, Mack, & Massara, 2006)

#### **2.2.2.6 Corner Sequence**

Corner Sequence is also used to represent non-slicing floorplan and is a P-admissible representation. Corner Sequence consists of two tuples that represent the packing sequence of the blocks and the corners which the block will be placed. Corner Sequence (CS)  $= \langle (S_1, D_1) (S_2, D_2) \dots (S_m, D_m) \rangle$  uses a packing sequence  $S$  of the modules well as the corresponding bends  $D$  formed by the modules to describe a compacted placement. Each two-tuple  $(S_i, D_i)$ ,  $1 \leq i \leq m$ , is referred as a term of the CS.

A module  $b_i$  is said to cover another module  $b_j$  if  $b_i$  is higher than  $b_j$  and their projections in the x-axis overlap, or  $b_i$  is right to  $b_j$  and their projections in the y-axis overlap (i.e.,  $y_j \leq y_i$ ,  $x_j > x_i$  and  $x_j < x_i + w_i$ , or if  $x_j \leq x_i$ ,  $y_j > y_i$  and  $y_j < y_i + h_i$ ). Here,  $x_i = x_i + w_i$  and  $y_i = y_i + h_i$ . Given an admissible placement (a left and bottom compacted

placement), firstly, pick the dummy modules  $b_s$  and  $b_t$ , and make  $R = \langle st \rangle$  for the two chosen modules. The module  $b_i$  on the bottom-left corner of  $P$  is picked (i.e.,  $S_1 = b_i$  and  $D_1 = [s, t]$ ) since it is the unique module at the bend of  $R$ , and the new  $R$  becomes  $\langle sit \rangle$ . When there is more than one module at bends, the left-most module that does not cover other unvisited modules is picked at the bends. Therefore, the module  $b_j$  at the bend  $[s, i]$  is picked if  $b_j$  exists and  $b_j$  does not cover the other unvisited module  $b_k$  at the bend  $[i, t]$ ; otherwise,  $b_k$  is picked. This process continues until no module is available. Based on the above procedure, there exists at least one module at a bend of the current  $R$  before all modules are chosen since the placement is compacted. Therefore, there exists a unique CS corresponding to a compacted placement.

Figures 2.20(a)–(h) show the process to build a CS from the placement  $P$  of Figure 2.20(a).  $R$  initially consists of  $s$  and  $t$ . Module  $a$ , at bottom-left corner is chosen first since it is the unique module at the bend of  $R$  ( $S_1 = a$ , and  $D_1 = [s, t]$ ). Figure 2.21(a) shows the resulting  $R$  (denoted by heavily shaded areas). Similarly, module  $b$  is chosen ( $S_2 = b$  and  $D_2 = [a, t]$ ) and the new  $R$  is shown in Figure 2.21(b). After module  $b_d$  in Figure 2.21(b) is chosen,  $a$  and  $b$  are removed from  $R$  since  $b_a \leq_x b_d$  and  $b_b \leq_y b_d$  (see Figure 2.21(c) for the new  $R$ ). As shown in Figure 2.21(d), there exist two modules  $b_f$  and  $b_c$  at bends. Although  $b_f$  is left to  $b_c$ , we pick  $b_c$  first since  $b_f$  covers  $b_c$ . This process repeats until no module is available, and the resulting CS is shown in Figures 2.21(i). (Lin, Chang, & Lin, Corner sequence - a P-admissible floorplan representation with a worst case linear-time packing scheme, 2003)



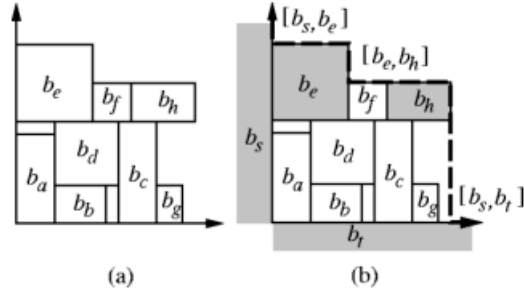


Figure 2.20(a) Placement, P (b) Contour R of P

The dynamic sequence packing (DSP) scheme is used to transform a CS into a placement. For DSP, a contour structure is maintained to place a new module. Let  $L$  be a doubly linked list that keeps modules in a contour. Given a CS, we can obtain the corresponding placement in  $O(m)$  time by inserting a node into  $L$  for each term in the CS, where  $m$  is the number of modules.  $L$  initially consists of  $n_s$  and  $n_t$  that denote dummy modules  $s$  and  $t$ , respectively. For each term  $(i, [j, k])$  in a CS, we insert a node  $n_i$  between  $n_j$  and  $n_k$  in  $L$  for module  $b_i$ , and assign the  $x$  ( $y$ ) coordinate of module  $b_i$  as  $x_j'$  ( $y_k'$ ). This corresponds to placing module  $b_i$  at the bend  $[j, k]$ . Then, those modules that are dominated by  $b_i$  in the  $x$  ( $y$ ) direction should be removed from  $R$ . This can be done by deleting the predecessor (successor)  $n_p$ 's of  $n_i$  in  $L$  if  $y_p$ 's ( $x_p$ 's) are smaller than  $y_i'$  ( $x_i'$ ). The process repeats until no term in the CS is available. Let  $W$  ( $H$ ) denote the width (height) of a chip.  $W = x_u'$  ( $H = y_v'$ ) if  $n_u$  ( $n_v$ ) is the node right before (behind)  $n_t$  ( $n_s$ ) in the final  $L$ .

Figure 2.22 gives an example of the packing scheme for the CS shown in Figure 2.22(a).  $L$  initially consists of  $n_s$  and  $n_t$ . We first insert a node  $n_a$  between  $n_s$  and  $n_t$  since  $S_1 = a$  and  $D_1 = [s, t]$ . The  $x$  ( $y$ ) coordinate of  $b_a$  is  $x_s'$  ( $y_t'$ ). Figure 2.22(b) shows the resulting placement and  $L$ . Similarly,  $n_b$  is inserted between  $n_a$  and  $n_t$  in  $L$  of Figure 2.22(b) since  $S_2 = b$  and  $D_2 = [a, t]$  (see Figure 2.33(c) for the resulting placement and  $L$ ). After we insert a node  $n_d$  between the two nodes  $n_a$  and  $n_b$  in  $L$  of Figure 7(c) for the third term  $(d, [a, b])$  in the CS, the predecessor  $n_a$  (successor  $n_b$ ) of  $n_d$  is deleted because

$y_a' \leq y_d'$  ( $x_b' \leq x_d'$ ) (see Figure 2.22(d)). The process repeats for all terms in the CS, and the resulting placement and L are shown in Figure 2.22(i). The width (height) of a chip is  $W = x_h'$  ( $H = y_e'$ ) since the node right before (behind)  $n_t$  ( $n_s$ ) is  $n_h$  ( $n_e$ ) in L. The DSP packing scheme packs modules correctly in  $O(m)$  time, where  $m$  is the number of modules.

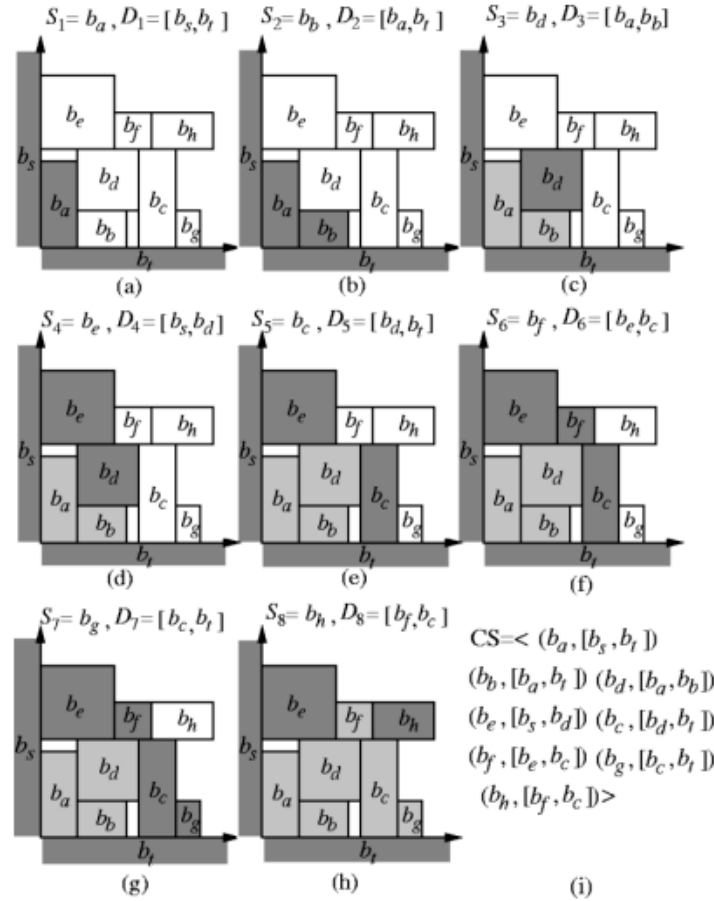


Figure 2.21 (a) – (h) The process to build a CS from placement (i) CS representation

The solution space of CS is bounded by  $(m!)^2$ , where  $m$  is the number of modules. It should be noted that, in addition to the number of modules, the solution space of CS also depends on the dimensions of the modules. The above theorem considers the worst case for CS—all modules appear in the contour all the time during packing. It is quite often that only part of the modules is in the contour. Hence, practical solution space of CS is smaller than  $(m!)^2$ .

$$CS = \langle (b_a, [b_s, b_t]) (b_b, [b_a, b_t]) (b_d, [b_a, b_b]) (b_e, [b_s, b_d]) \\ (b_c, [b_d, b_t]) (b_f, [b_e, b_c]) (b_g, [b_c, b_t]) (b_h, [b_f, b_c]) \rangle$$

(a)

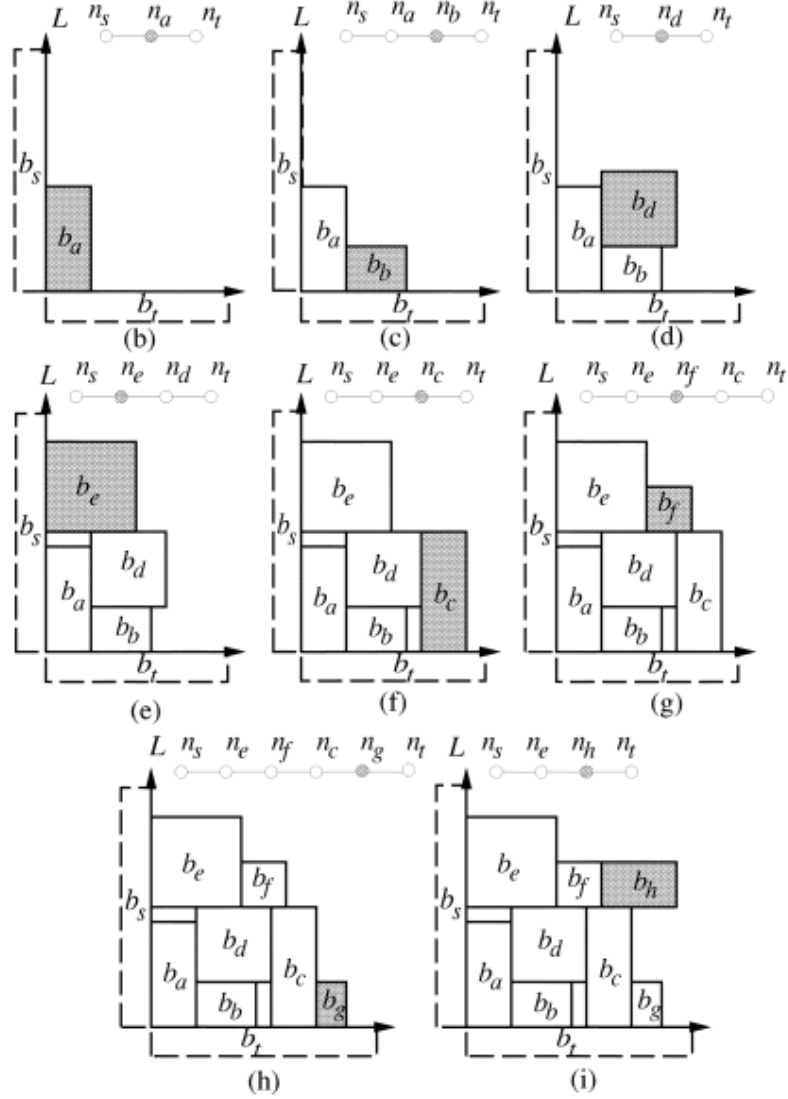


Figure 2.22 (b) – (i) DSP packing scheme for CS in (a), where CS =  $\{(a, [s, t]) (b, [a, t]) (d, [a, b]) (e, [s, d]) (c, [d, t]) (f, [e, c]) (g, [c, t]) (h, [f, c])\}$

## 2.3 Optimization Algorithms

Several approaches have been used to for the floorplanning problem such as constructive, iterative and knowledge based. Constructive algorithms attempt to build a feasible solution by starting from a seed module. After that, the modules are selected one at a time and added to the partial floorplan. This process continues until all modules have been selected. This type of algorithm includes cluster growth, partitioning and slicing, connectivity clustering, mathematical programming, and rectangular dualization.

Iterative techniques start from an initial floorplan. Then this floorplan undergoes a series of perturbations until a feasible floorplan is obtained or no more improvements can be achieved. Typical iterative techniques which have been successfully applied to floorplanning are simulated annealing, force directed interchange/relaxation, and genetic algorithm.

The knowledge-based approach has been applied to several design automation problems including cell generation and layout, circuit extraction, routing, and floorplanning. In this approach, a knowledge expert system is implemented which consists of three basic elements:

1. Knowledge base that describes floorplan problem and its current state
2. Rules for data manipulation to progress toward a solution
3. An inference engine controlling the application of the rules

The type of approach for optimization that will be discussed in the following will be the iterative technique. Hence, more iterative techniques are reviewed before the project is being done. (Sait & Youssef, 1999)

### 2.3.1 Simulated Annealing

Simulated annealing algorithm is starting with an initial solution and armed with adequate perturbation and evaluation functions. The algorithm performs a stochastic local search of the state space. The parameter that controls the accepted rate is controlled by temperature ( $T$ ). The probability of acceptance of decreases as temperature decreases.

The core of this algorithm is the *Metropolis* procedure, which simulates the metal's annealing process at a given temperature  $T$ . The *Metropolis* procedure receives as input the current temperature  $T$ , and the current solution  $CurS$ . *Metropolis* procedure must also be provided with the value  $M$ , which is the amount of time for which annealing must be applied at temperature  $T$ . Temperature is initialized to  $T_0$  at the beginning, and is slowly reduced to achieve cooling. It uses a procedure *perturb* to generate a new solution  $NewS$ . If the cost of the  $NewS$  is better than the cost of the  $CurS$ , then the  $NewS$  is accepted, and we do so by setting  $CurS = NewS$ . If the cost of the  $NewS$  is better than the best solution  $BestS$ , then we will replace  $BestS$  with  $News$ . If the  $NewS$  has a lower cost in comparison to the  $CurS$ , *Metropolis* will accept the  $NewS$  on a probabilistic basis. If a random number, which is generated range from 0 to 1, is smaller than  $\Delta cost/T$ , where  $\Delta cost = cost(NewS) - cost(CurS)$ , and  $T$  is the current temperature, the inferior solution is accepted. The simulated annealing algorithm needs to start from a high temperature. However, if this initial value of  $T$  is too high, it will take a long processing time. The initial temperature is usually set as  $T_0 = avg/\log(P)$ . The stopping criterion is when 0.1 or reject ratio  $T > 0.95$ . (Xu & Li, 2008)

Chen and Chang suggest a study on two types of modern floorplanning problems, which are fixed-outline floorplanning and bus driven floorplanning. In their paper, they use B\*-tree floorplan representation based on fast three-stage simulated

annealing scheme called Fast-SA. Fast-SA can dynamically change the weights in the cost function to optimize the area and also wirelength with various aspect ratios. Both soft block and hard block can be optimized using Fast-SA. This method is an improvement from SA which can increase the speed for optimization. (Chen & Chang, 2006)

Chen, Zhu and Ali suggested a hybrid simulated annealing (HSA) for non-slicing floorplan. HSA uses a new greedy method to construct an initial B\*-tree which is a new operation on the B\*-tree to explore the search space and a novel bias search strategy to balance global exploration and local exploitation. Hence, HSA can give quicker optimal or nearly optimal solutions compared to SA. (Chen, Zhu, & Ali, A Hybrid Simulated Annealing Algorithm for Nonslicing VLSI Floorplanning, 2010)

### **2.3.2 Genetic Algorithm**

GA was first proposed by Holland in 1975. In nature only fittest individuals survive and reproduce, a natural phenomenon known as “the survival of the fittest”. GA mimics the natural evolution process by suppressing inferior genotypes and breeding offspring from superior population members. The cyclic process is continued for several generations and the best member is selected. The performance of a GA can be drastically improved by using elitism, which ensures that best-known solution is preserved and passed on to future generations.

The algorithm randomly generates a set of population, which is called the first generation. The population will consist of various set of strings which are known as genes. These strings of genes make up chromosomes. The chromosomes represent the solution of the optimization problem. Each chromosome will be evaluated at every iteration or generation. The evaluation will determine the fitness of the gene. Basing on the fitness, individuals called parents are selected from the population. The individual

with higher fitness has more probability of being selected. After that, some genetic operators will be done and the output will be called offsprings. The genetic operator combines both the parents' features. The more common operators are such as cross over and mutation. Then the offspring will then be the second generation and another set of generation will be randomly produced including the offspring from the parents. This will go on until the optimization is being stopped. (Debarshi & Manikas, 2007)

A pseudo code for our elitist GA is given below:

**Begin**

**input:** Block Dimensions,  $w \times h \forall i \in \Pi_{ii}$ /\*MCNC

Benchmark files\*/

**output:** Block Coordinates,  $x \times y \forall i \in \Pi_{ii}$ /\*Layout

Files\*/

- 1: Initialize population by assigning random pair of permutations.
  - 2: Create empty external population  $P_{ext}$  with max. size  $N_{ext}$  to store the best members.
  - 3: if number of members in  $P_{ext} > N$
  - 4: Delete the worst member in  $P_{ext}$
  - 5: Select members from the current population
  - 6: Apply crossover
  - 7: Apply mutation
  - 8: Replace a small proportion of the new population by random members of  $P_{ext}$ .
  - 9: if (no of generations < constant1 or fitness of best member in  $P_{ext}$  < constant2)
  - 10: Go to step 3.
- end

Nakaya proposed to use an adaptive genetic algorithm to solve floorplanning problem in VLSI layout design. They used the sequence pair as representation and is adopted the coding scheme of each chromosome. Analysis of new operator for GA is explored to improve results. The proposed GA has an adaptive strategy which dynamically selects an appropriate genetic operator during the GA execution depending on the state of an individual. (Nakaya, Koide, & Wakabayashi, 2000)

Chen and Zhu suggest a hybrid genetic algorithm (HGA) which is modified from GA to solve the non-slicing and hard-module of VLSI floorplanning problem in

order to improve the optimization results for floorplanning optimization. HGA uses an effective genetic search method to exploit information in the search region. (Chen & Zhu, A hybrid Genetic Algorithm for VLSI Floorplanning, 2010)

### **2.3.3 Cross Entropy Method**

The Cross Entropy (CE) method is developed based on the cross entropy distance or also known as the Kullback-Leibler) distance. This is a fundamental concept of modern information theory. This method is motivated by an adaptive algorithm to estimate probabilities in rare events which involve minimization. CE can be used for in estimating probabilities of rare events in a complex stochastic network that needs minimization. Besides that, it can also be used to solve difficult combinatorial optimization and continuous multi-extremal problems. This could be done by translating deterministic optimization into stochastic estimation. CE involves iterative procedure which can be divided into two phases which are:

1. Generation of random data such as trajectories and vectors from a specific mechanism.
2. Updating the parameters of the random mechanism based on the performance of the data in order to obtain better samples in the next iteration.

The iteration for CE will stop once it reaches the stopping criteria. The stopping criterion is selected depending on the noise of the results obtained. (Rubinstein & Kroese, The Cross-Entropy Method: A unified approach to Combinatorial Optimization, Monte-Carlo Simulation and Machine Learning, 2004)



## CHAPTER 3. METHODOLOGY

This chapter discusses the method developed in this project. There were 3 different methods of floorplanning optimization used. The first method uses Dot Model as representation and Genetic Algorithm for optimization. The second method uses Corner Bottom Left List as representation and Genetic Algorithm for optimization. The third method uses Corner Bottom Left List as representation which is developed for this project together with a modified Cross Entropy method for optimization. All these methods are focused on optimizing the floorplan area to achieve the minimum deadspace.

### 3.1 Dot Model as Representation and Genetic Algorithm as Optimization Algorithm

This method uses the Dot Model to represent the floorplan and Genetic Algorithm as the optimization algorithm. Dot Model (DM) is a topological representation where it uses the relative position with reference to x-axis and y-axis in order to place the rectangle block position in the floorplan. Dot Model is developed based on the position where rectangle blocks are placed from the left bottom corner till all the blocks are exhausted. This is done based on selecting the block and placing the block according to the position and orientation that is given with reference to the previous block number.

DM consists of three tuples that denote the packing sequence of modules, the reference block number and the orientation of the block for the module. There are 4 orientations which were used which are right, right with rotation, top and top with rotation. The rotation is  $90^\circ$  for both right with rotation and top with rotation.

Genetic Algorithm has been widely used for optimization for floorplan. In this work, we try to combine dot model with Genetic Algorithm. Genetic Algorithm is

used global search for optimization. The GA that is used in this method uses the representation of DM to optimize the floorplan. The representation of DM can be used in GA by modifying the chromosomes of the parents and child so that it can optimize the encoded DM to obtain optimum results.

### 3.1.1 Dot Model

Dot Model is developed by referring to topological representation and then is encoded in to numerical representation. DM has 3 tuples  $(B_i, R_i, P_i)$ . The first tuple,  $B_i$ , represents the block number which will be taken to be placed into the floorplan. The second tuple,  $R_i$ , represents the block number which is referred to place the block. The third tuple,  $P_i$  represents the orientation and rotation of the position of the block number,  $B_i$  that is placed with reference to the second tuple,  $R_i$ .

This part will discuss on the representation of DM.  $DM = \{(B_1, R_1, P_1) (B_2, R_2, P_2) \dots (B_n, R_n, P_n) \}$ . B represents the sequence of the block number of m blocks that is selected to be placed on the floorplan according to the R and P. The three tuples are referred as  $(B_i, R_i, P_i)$ ,  $1 \leq i \leq m$  as the term for DM. Next, we will discuss on how the DM is placed into placement.

#### 3.1.1.1 DM to placement

First of all, a string of solution,  $(B_i, R_i, P_i)$  is checked and analysed. We need to check whether the DM solution is valid or not. This means that the reference block number,  $R_i$  must come after the block sequence,  $B_i$ . If the reference block number,  $R_i$  comes before the block sequence,  $B_i$ , then  $(B_i, R_i, P_i)$  must be pushed down to  $(B_{i+a}, R_{i+a}, P_{i+a})$  where a represents the number of times it must be pushed down until the block  $R_i$  is found at  $B_{i+a-1}$ . This step must be done until all the reference block numbers,  $R_i$  appears after the block sequence,  $B_i$  so that the DM solution string is valid. If after

completing checking the solution string of DM and there are still invalid DM solutions, the block will be placed automatically using deterministic method in order to obtain the minimum dead space area for that particular block. Figure 3.1 is an example of a solution string that is randomly generated by GA,  $DM = (B,R,P)$  and how it is shifted in order to make  $(B_2,R_2,P_2)$  to be valid. This shifting will be taken place until all the reference block, R appears after the sequence block and the end solution that will be used for placement in DM is shown in Figure 3.2. The circled tuples at Figure 3.2 shows that the tuples that is not valid and needs to use deterministic method in order to obtain the minimum deadspace area for the blocks where it does not depend on GA for optimization.

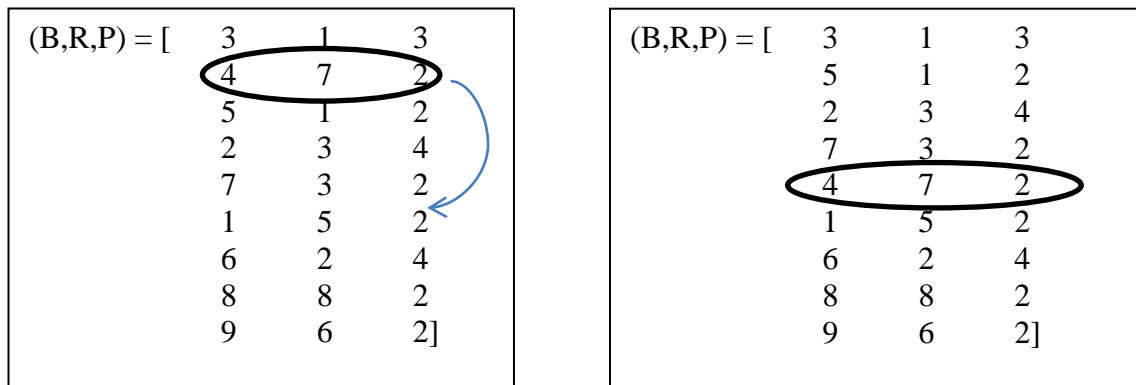


Figure 3.1 DM solution string and shifting of the solution string

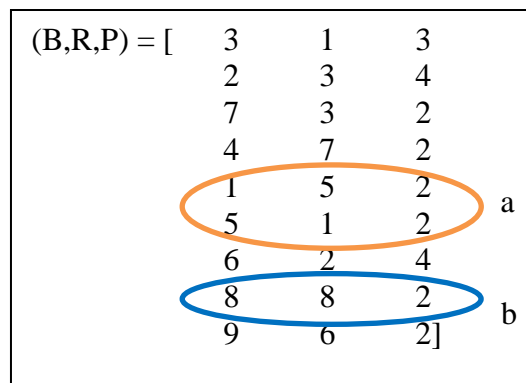


Figure 3.2 Solution string that is used for placement

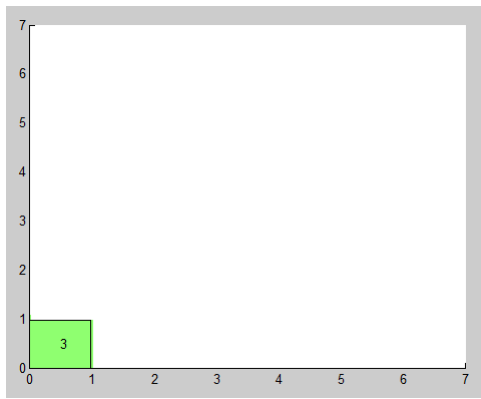
After validating the solution string, the placement can be done according to the solution string. First of all, the first block is placed on the bottom left corner of a plane

with axis x and axis y.  $R_1$  can be ignored as the first block does not require any reference block.  $P_1$  will be used to check the rotation of the first block. Generally, there are four types of position that are used for DM for P. The positions that are used are:

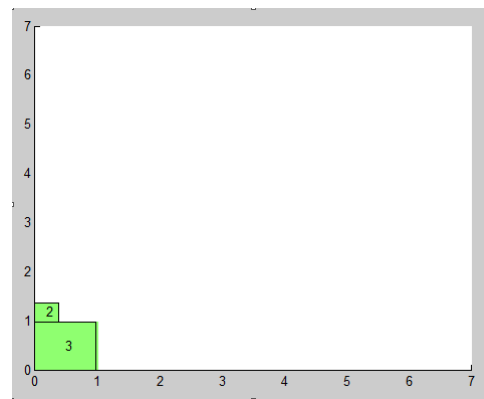
- 1 – Place block, B on the right of the reference block, R without rotation
- 2 – Place block, B on the right of the reference block, R with  $90^{\circ}$  rotation
- 3 – Place block, B on top of the reference block, R without rotation
- 4 – Place block, B on top of the reference block, R with  $90^{\circ}$  rotation

However, the first row of the solution string does not need the reference block and hence 1 and 3 will represent that block,  $B_1$  does not have rotation and 2 and 4 will represent that block,  $B_1$  needs to be rotated  $90^{\circ}$ .

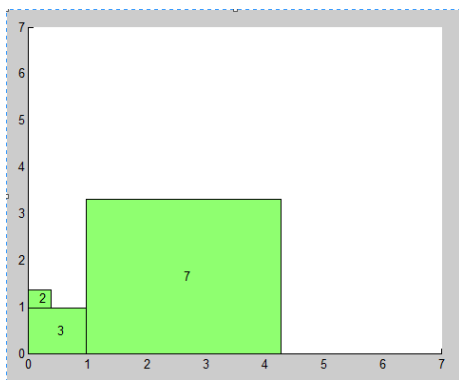
For the following rows,  $(B_2, R_2, P_2), \dots, (B_n, R_n, P_n)$ , the positions for P needs to be used. After placing the first block, the subsequent block number will be placed with reference to R according to the position of P. The placement will be shifted so that no two blocks will overlap one another. If there is an invalid row, the block will be placed at the origin position and it will be placed again in another part of the floorplan to give the minimum dead space area so that there will be no overlapping that will happen in the floorplan. The placement of the blocks corresponds to one another and depends on the previous block which has been placed into the placement. It is important to shift and place the block so that it will give a more compacted floorplan. Figure 3.3 will show how the placement of the blocks according to the DM solution from figure 3.1 that is generated by GA.



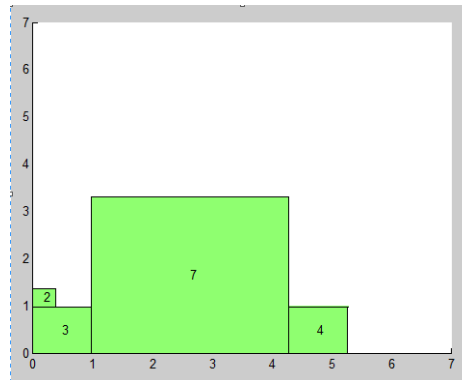
i) Placement of first block



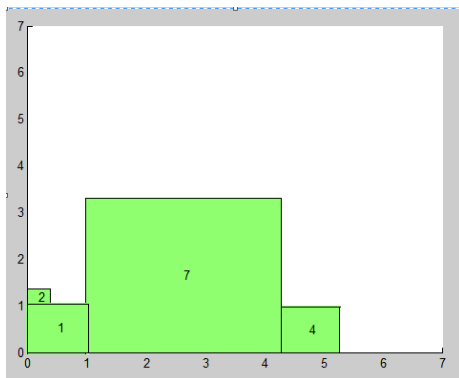
ii) Placement of second block



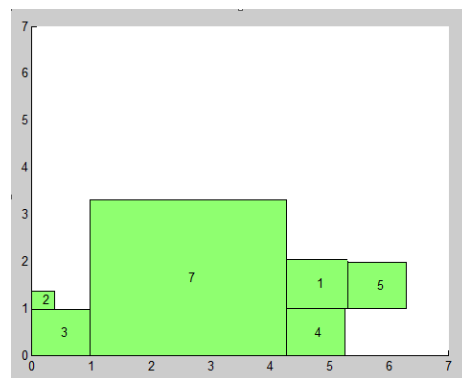
iii) Placement of third block



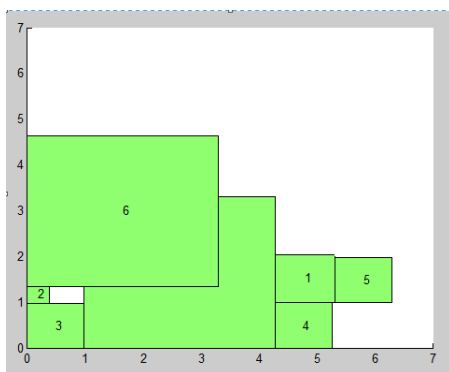
iv) Placement of fourth block



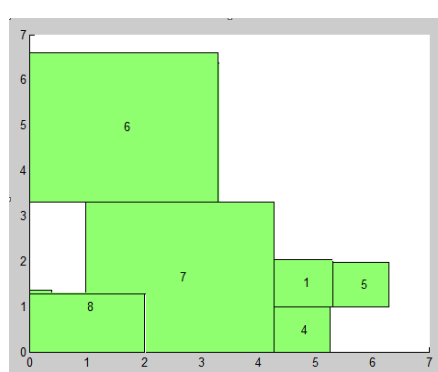
v) Placement of fifth block



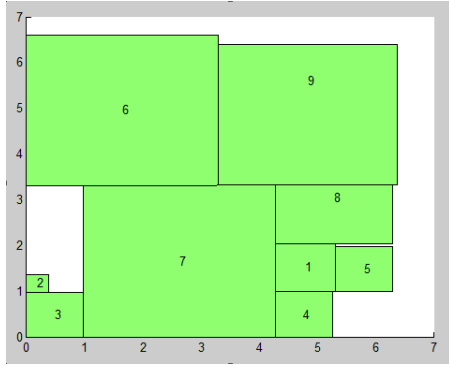
vi) Placement of sixth block



vii) Placement of seventh block



viii) Placement of eighth block



ix) Placement of ninth block

Figure 3.3 Placement from DM to floorplan

According to the given example, we can observe that the DM is not suitable to do direct placement. Hence, it is needed to shift the sequences of the tuples so that the DM can be used for placement. After shifting, we get the solution string as shown in Figure 3.1. The first row of DM = (3,1,3). Hence, block 3 is placed at the bottom corner left of the plane and no rotation is done for block 3. This is shown in Figure 3.3(i). After that, we get the second row of DM = (2,3,4). From here, we place block 2 with reference to block 3 according to the orientation stated which is 4, meaning that we place block 2 on top of block 3 and make a rotation of  $90^\circ$  for block 2. This is shown in Figure 3.3(ii). Next, we check the third row of DM = (7,3,2). This means that we will place block 7 with reference to block 3. We then place block 7 on the right of block 3 and rotate block 7. This is shown in Figure 3.3(iii). After that, we check the fourth row of DM = (4,7,2). This means that we need to place block 4 with reference to block 7. The P is 2, meaning that block 4 is placed on the right of block 7 and block 4 is rotated. This is shown in Figure 3.3(iv). Then, we check the fifth row, DM = (1,5,2). It is observed that this row is not a valid placement as there is no block 5 placed on the plane. Hence, block 1 is first placed on the origin and then is shifted so that it will give a minimum dead space area and does not overlap with other block. As we can see in Figure 3.3(v), block 1 is placed at the origin of the plane as this row is not a valid placement. Hence, it needs to be shifted. Block 1 is shifted to the right and placed on

top of block 4. This can be seen in Figure 3.3(v). Next, we check the sixth row, DM = (5,1,2). We will place the block 5 referring to block 1. Hence, block 5 is placed on the right of block 1 and is rotated 90°. This can be seen in Figure 3.3(vi). After this, we check the seventh row, DM = (6,2,4). Hence, we will place block 6 referring to block 4. Block 6 will be placed on the top of block 2 and is rotated 90°. This can be seen in Figure 3.3(vii). Then, we check the eighth row, DM = (8,8,2). This is also not a valid row as block 8 cannot be referred to block 8. Hence, block 8 is placed at the origin as shown in Figure 3.3(viii). Then it is shifted to the right and placed on top of block 1. This is determined by the minimum deadspace that is placed on that location. This can be seen in Figure 3.3(ix). Finally the last row, DM = (9,6,2). This shows that block 9 is placed with reference to block 6 and is placed on the right of block 6 by rotating 90°. This can be seen in Figure 3.3(ix).

Through DM, we are able to calculate the deadspace area of the floorplan. The area is calculated in terms of percentage. This is calculated by using the upper boundary of the floorplan for both x and y axis and minus the total area of the blocks. Below show the equation that is used to calculate the deadspace area.

$$A = \frac{X_{boundary} * Y_{boundary} - \sum_{i=1}^n X_i * Y_i}{\sum_{i=1}^n X_i * Y_i} \quad (1)$$

Where  $X_{boundary}$  = right most block boundary of x-axis,  $Y_{boundary}$  = top most block boundary of y-axis,  $X_i$  = width of  $i$ -th block,  $Y_i$  = height of  $i$ -th block. The example above shows the concept of dot model which is used to represent a floorplan. In order to optimize the floorplan, we need to use the representation of DM and placed into GA so that optimization can be done to obtain optimum placement. The next section will discuss about GA and how GA is used together with DM in order to optimize a floorplan.

### 3.1.2 Genetic Algorithm

Genetic algorithm or also known as GA is a heuristic search technique which is adapted from the natural process of evolution so that an optimum gene can be obtained in order to achieve survival of the fittest. GA utilizes the theory of evolution according to the biological way by using methods like ‘crossover’, ‘mutation’ and selection in order to obtain better children in the next generation. (Sait & Youssef, 1999)

In order to use GA for optimization, we need to have two requirements which are the chromosomes and also the fitness function. A string of chromosome representation shows the characteristics of the gene. These chromosomes usually formed by bit string where they represent the characteristics of the solution in the chromosomes. In optimization of floorplan, the chromosome used represents the encoded DM where the DM can be decoded and form the floorplan which shows the packing and placement of the floorplan. This shows that each chromosome holds information of the floorplan that is needed for the placement. The fitness function measures the quality of the chromosome. Fitness function is important to determine that the solution fits the criterion and constraint of the situation. This is important to optimize the GA as GA process is based on the fitness of the chromosome which will be brought to the next generation. According to GA theory, evolution can give better generation. Hence we should obtain better solution from every subsequent generation as the best selected solution will be used for the next generation. Those that do not meet the fitness criterion will be eliminated. (Mitchell, 1999)

GA consists of 3 operators which are selection, crossover and mutation. For the selection operator, this is done by selecting the top fittest chromosomes which will be used for the next generation and the other chromosomes that do not fit the criterion will be eliminated. Hence, only the fitter chromosomes will be able to survive in the next



generation. For the crossover operator, a random selected locus will be chosen and the sub-sequence before and after the locus will be exchanged between two chromosomes to produce two new children. Usually, the fitter chromosomes are chosen to do the crossover operations so that better children will be produced in the next generation. For the mutation operator, one of the chromosomes will be chosen and one of the bits in the chromosome will be flipped or will be randomly located inside the chromosomes to change the characteristics of the chromosome and hence changes the fitness of the chromosomes. Figure 3.4 will show the crossover operator and Figure 3.5 will show the Mutation operator.

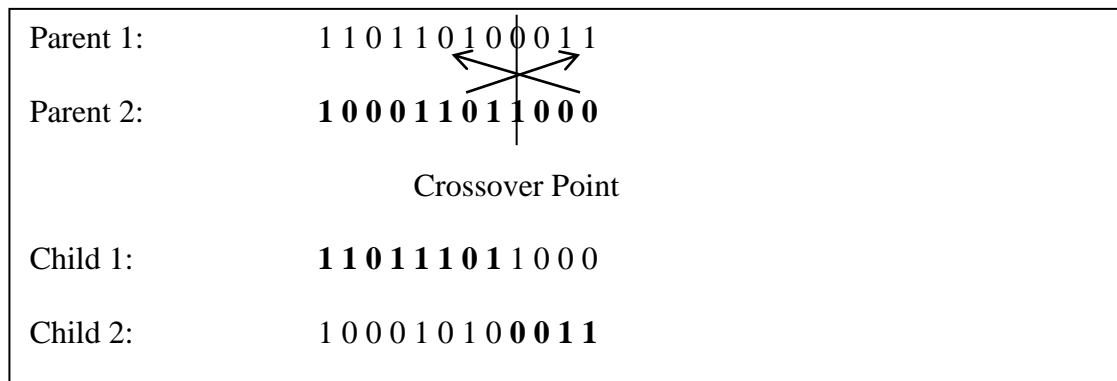


Figure 3.4 Crossover Operator

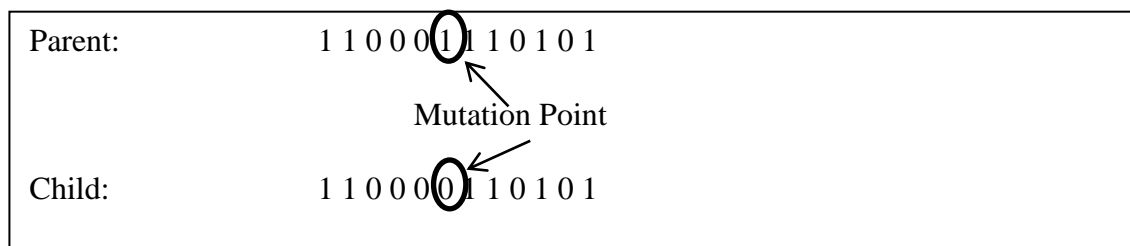


Figure 3.5 Mutation Operator

Allowing genetic operations increase the solution space of the generation and chances are performing this operator can improve the fitness of the chromosomes. However, the pool of solution will become closer to the real solution as the less fit chromosomes are discarded and the more fit solutions are maintained until the end of the GA optimized solution.

Figure 3.6 is the general pseudocode of a genetic algorithm and Figure 3.7 is the general flow chart of a genetic algorithm. The implementation of the GA will be discussed in the following section.

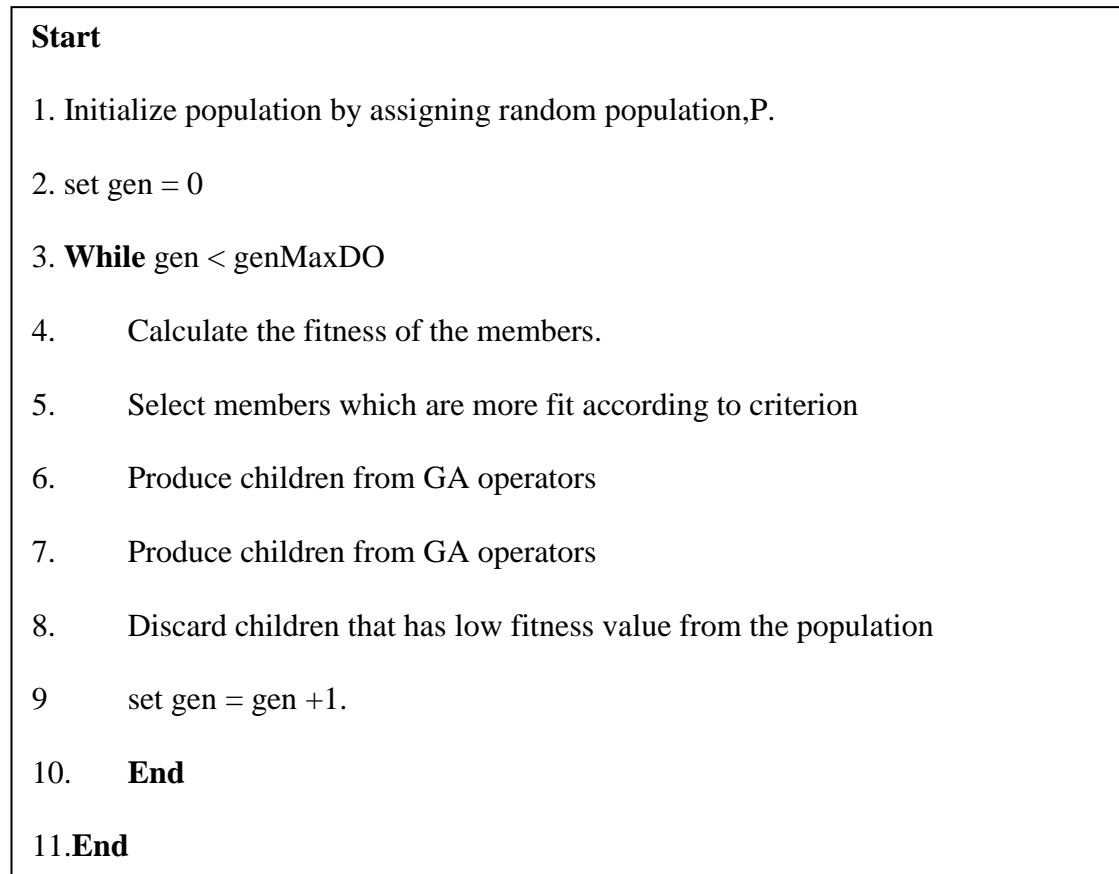


Figure 3.6 GA pseudocode

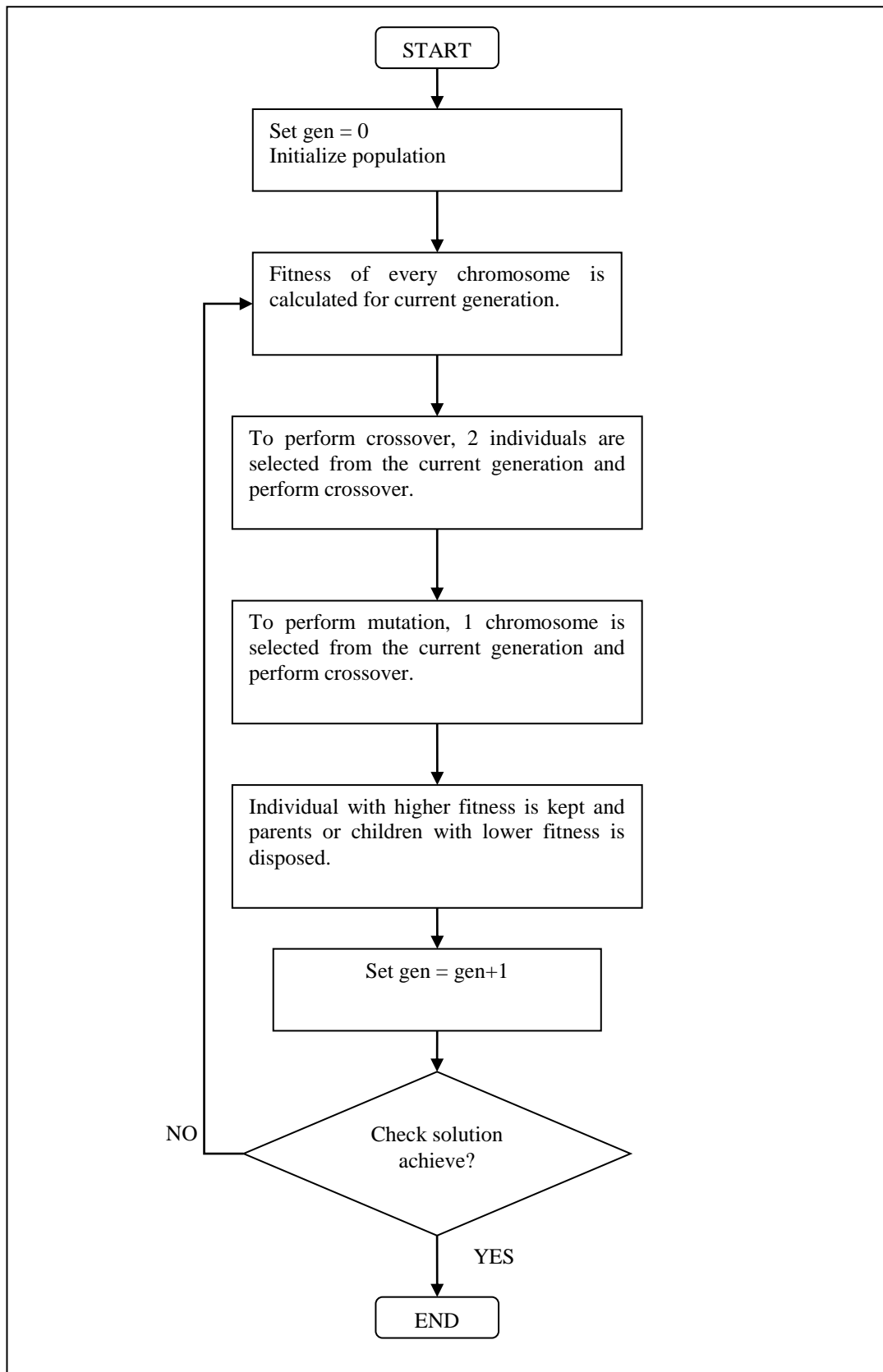


Figure 3.7 GA flowchart

### 3.1.2.1 Implementation of GA with DM

This GA source code is an open source code was developed by C. R. Houck, J. Joines, and M. Kay distributed under GNU General Public License as published by the Free Software Foundation and is available at <http://www.ise.ncsu.edu/mirage/GAToolBox/gaot/gaotindex.html>. From this source code, a slight modification is done in order to use for DM. (Houck, Joines, & Kay)

DM consists of three tuples which gives  $(B_i, R_i, P_i)$ . First of all, to use GA, we need to model the chromosomes so that optimization using GA can be done. Hence, the chromosome for DM is modelled as in Figure 3.8.

$$\text{Chromosome} = \{(B_1, B_2, \dots, B_n) (R_1, R_2, \dots, R_n) (P_1, P_2, \dots, P_n)\}$$

Figure 3.8 Chromosome Model

B represents the block number. These numbers must be permuted numbers as one block can only be placed once in the floorplan. This is obtained by using random permutation in the matlab function. R represents the reference block. This block can be repeated and is replaceable. This means that we can refer to the same block more than once. In order to generate these random variables, the formula in Figure 3.9 is used.

$$R = \text{round}(\text{rand}(b_1, \dots, b_n) * (n-1) + 1)$$

Figure 3.9 Random Variables generation with repeated numbers

Finally, P represents the position and orientation of the block B with reference to block R. This representation needs to be generated from 1 to 4 randomly. In order to generate these random variables, the formula in Figure 3.10 is used.

$$R = \text{round}(\text{rand}(b_1, \dots, b_n) * 3 + 1)$$

Figure 3.10 Random Variables from 1 to 4 is generated

After modelling the chromosomes, the initialization of population according is carried out as the random generation stated earlier. The size of population used are determined by the number of blocks in the floorplan. When the number of blocks to be placed in floorplan increases, the number of population needs to be increased. Next after the initializing the initial population, the population is evaluated to obtain the fitness of the chromosomes in the population. The chromosomes are then arranged based on their fitness where the top being the fittest and the bottom is the worst. The fitness of the chromosomes are determined through DM where DM gives the deadspace area of the floorplan for every solution string. After arranging the population, the top quantile is selected to be brought to the next population. In this new population, a few chromosomes are selected to do the GA operators such as Crossover and also Mutation.

There are many types of crossover operators. But the crossover operators that are used in this work are arithmetic crossover, heuristic crossover and also simple crossover for the non-order based representation. The crossovers that are used for the ordered based representations are the cyclic crossover, order-based crossover, single point crossover and partial mapping crossover. Below are the explanations for the ordered based crossover function that are used in this work:

a) Cyclic crossover

Cyclic crossover uses two parents, P1 and P2 to perform the crossover to produce 2 children. The cyclic crossover copies the genes from the parent chromosome to the child chromosome in a cyclic manner as shown in Figure

3.11

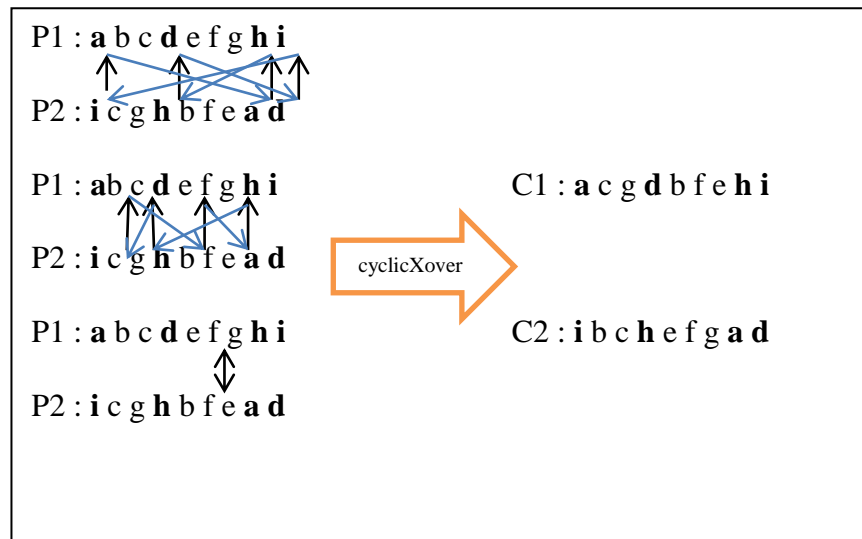


Figure 3.11 Cyclic crossover

#### b) Uniform crossover

Uniform crossover uses two parents, P1 and P2 to perform uniform crossover for a permuted string. The genes are treated independently and is randomly decided from which parent that the child will inherit the gene. For uniform crossover, a mask usually is generated and the crossover is based on the mask. This can reduce the bias which happens in single point crossover. Figure 3.12 shows how a uniform crossover operation is taken place.

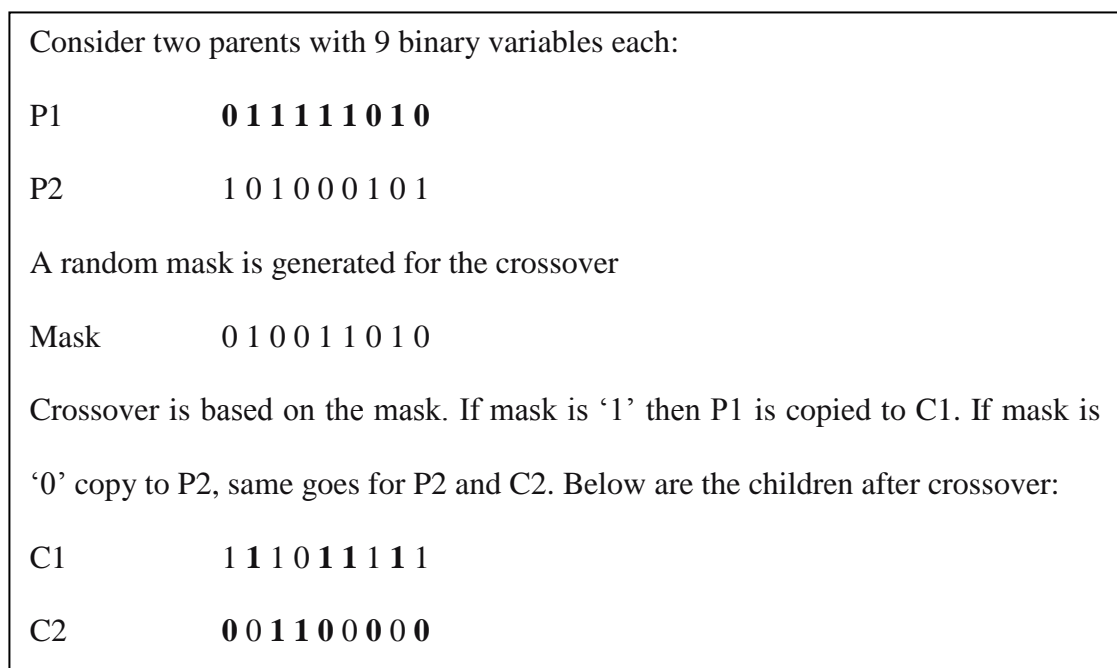


Figure 3.12 Uniform crossover

c) Partial mapping crossover

This crossover function also uses 2 parents, P1 and P2 and perform partial mapping on the chromosome. The steps for partial mapping crossover are shown in Figure 3.13.

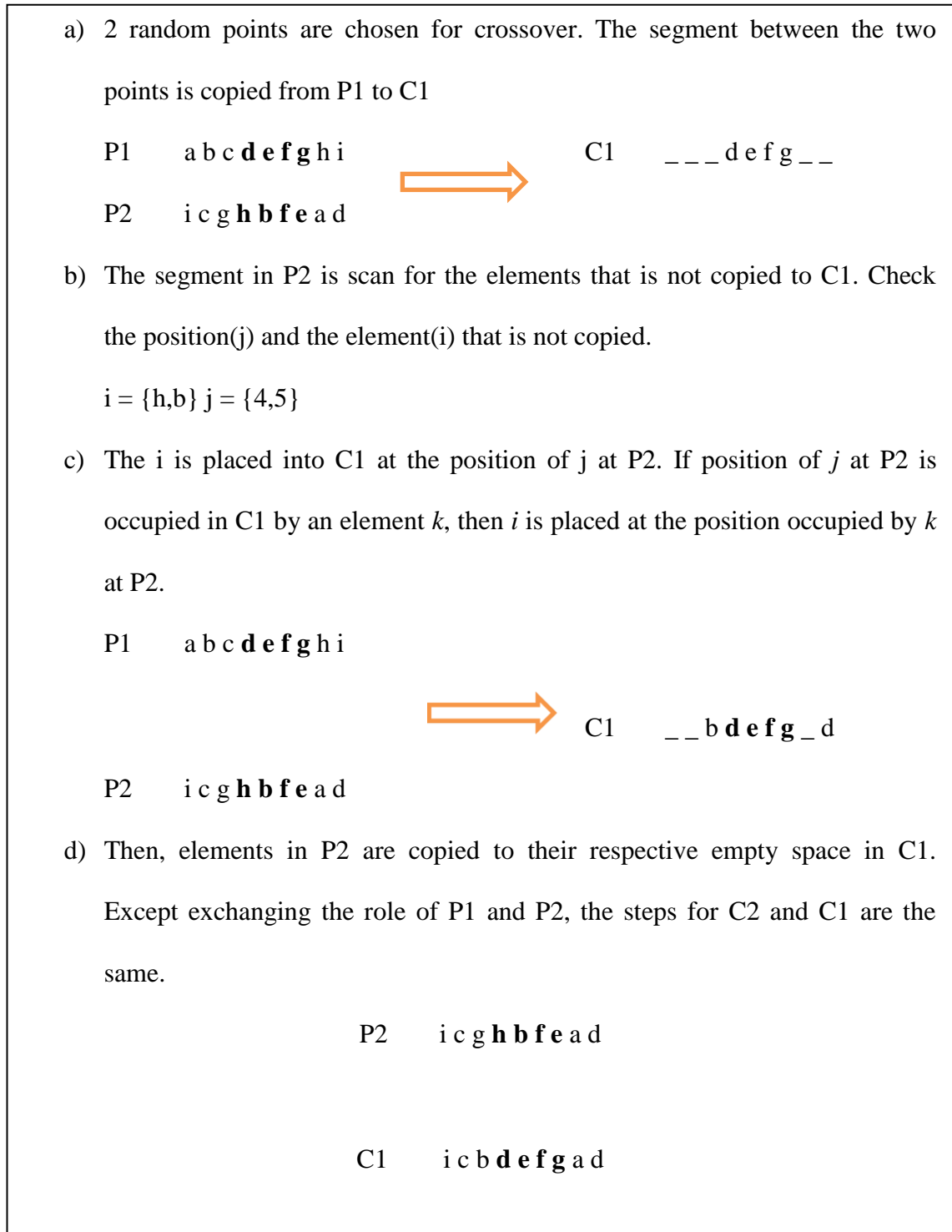


Figure 3.13 Partially mapped crossover

d) order-based crossover

This crossover operator uses 2 parents, P1 and P2 and is the same as partialmap crossover at the initial approach where it copies a segment from 1 parent. However, the following step differs in a way that the rest of the unused element in P2 is copied into C1 in the order as in P2. This will enable information from the relative sequence of element is transferred to the child. Figure 3.14 shows how an order-based crossover is done.

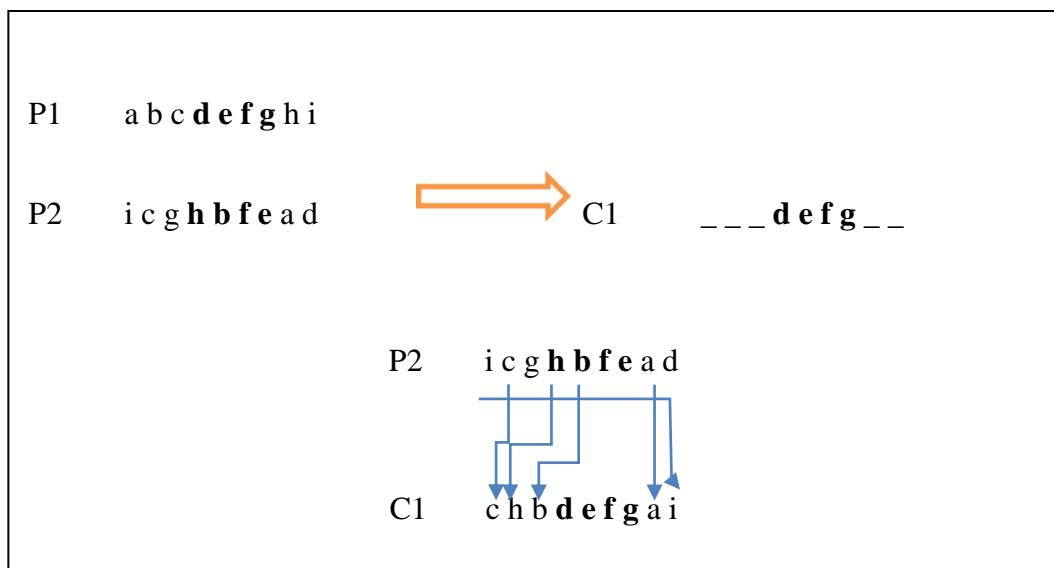


Figure 3.14 Order-based crossover

e) single point crossover

Single point crossover will select a random point and copy the segment from beginning to the point from the parent to the child. The remaining segment of the child will be taken from another parent except for the elements that has been copied. Figure 3.15 shows how a single point crossover is done

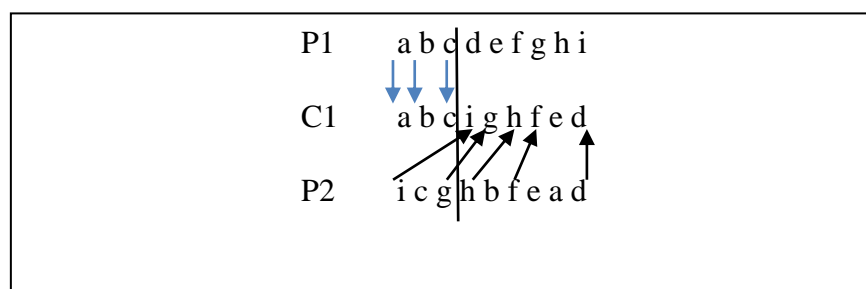


Figure 3.15 Single point crossover



f) linear order crossover

Linear order crossover is modified from order crossover. Linear order crossover gives the absolute position of the order for the elements in the chromosome.

Figure 3.16 shows how linear order crossover taken place.

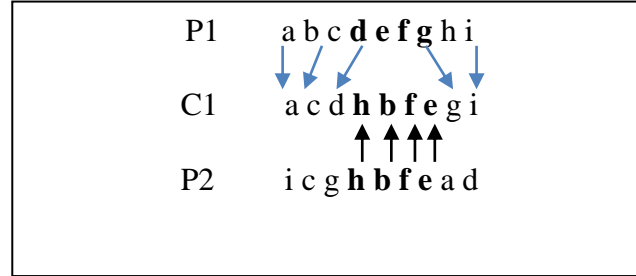


Figure 3.16 Linear order crossover

Besides order-based sequence, we also need to use the floating point representation for DM. Hence, we need to use the crossover operators for floating point representation. The crossovers that are used in this project are arithmetic crossover, heuristic crossover and simple crossover. Below are the explanations on the floating point representation crossover.

a) Arithmetic crossover

Arithmetic crossover uses two parents P1, P2 to perform an interpolation along the line formed by the two parents. This will form a mix of the 2 parents at a certain ratio. Figure 3.17 shows how the arithmetic crossover is formed.

$$C1 = P1 * a + P2 * (1-a);$$

$$C2 = P1 * (1-a) + P2 * a;$$

Figure 3.17 Arithmetic Crossover equation

b) Heuristic crossover

Heuristic crossover uses two parents P1, P2 to perform an extrapolation along the line formed by the two parents in the outward direction of the better parent. Heuristic crossover uses the fitness function of the parent chromosome to

determine the direction of search. Figure 3.183 shows the heuristic crossover equation.

$$C1 = \text{BestParent} + r * (\text{BestParent} - \text{WorstParent})$$

$$C2 = \text{BestParent}$$

Figure 3.18 Heuristic crossover

c) simpleXover

Simple crossover uses two parents P1, P2 to perform a simple single point crossover at a random point. Figure 3.19 shows the simple crossover function.

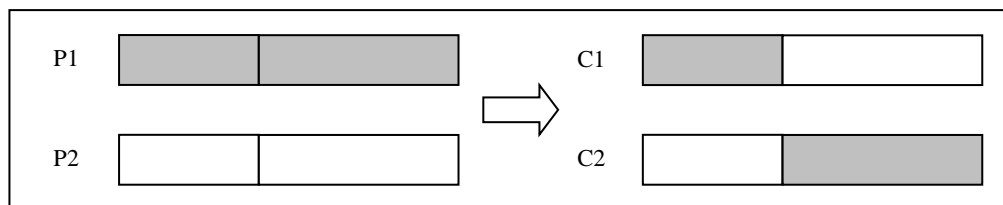


Figure 3.19 Simple crossover

After the crossover operator, a few of the chromosomes from the population is picked to do mutation operator. There are 5 types of mutation operator for order based part of the chromosomes which are used in this project. Below are the explanations for the mutation operator:

a) inversion Mutation

Inversion mutation select two random points between the chromosome string and the selected points are cut and inverts the bits or permutation of a chromosome string. Figure 3.20 shows how inversion mutation is done.

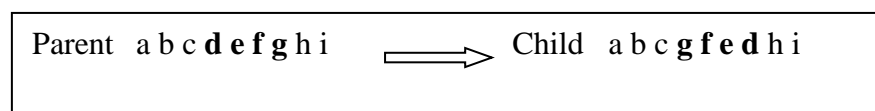


Figure 3.20 Inversion mutation

b) swap Mutation

Swap mutation chose two random genes in a permutation string and the two genes will exchange positions. This is shown in Figure 3.21.

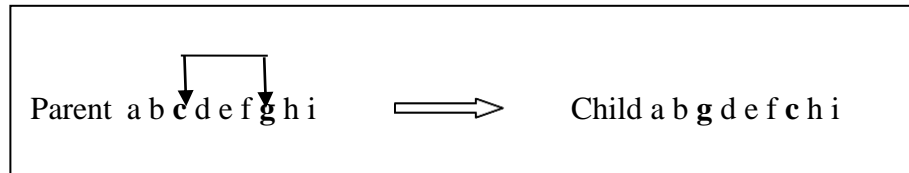


Figure 3.21 Swap mutation

c) adjswapMutation

Adjacent swap mutation swap two adjacent genes in a permutation string at a random selected point. This is shown in Figure 3.22

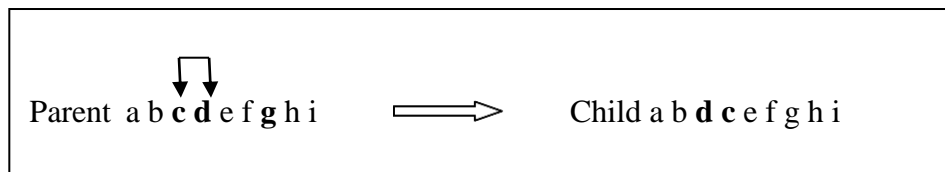


Figure 3.22 adjacent swap mutation

d) threeswapMutation

Three-swap mutation performs a three way swap of three randomly chosen genes in a permutation string. This is shown in Figure 3.23

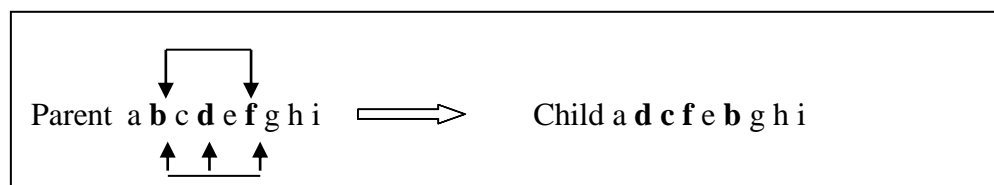


Figure 3.23 three swap mutation

e) shiftMutation

Shift mutation displaces one random gene in a permutation string to another position. This is shown in Figure 3.24.

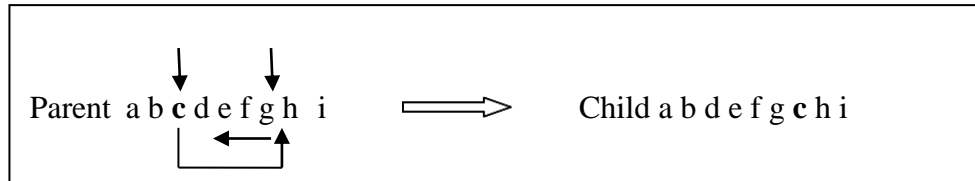


Figure 3.24 Shift mutation

Besides mutation for order-based number, mutation also is done for the floating point representation. There are four types of mutation which are used in this project for the floating point mutation which are the boundary mutation, multi non-uniform mutation and also the uniform mutation. Below are the explanations for these mutations that are used in this project.

a) Boundary Mutation

Boundary Mutation will change one of the parameters of the parents by randomly changing it to upper or lower boundary.

b) Multi non-uniform Mutation

Multi non-uniform mutation will change all the parameters in the parents based on a non-uniform probability distribution. This uses the Gaussian distribution that starts wide and is narrowed down in to a point distribution as the current generation approaches maximum generation.

c) Non uniform Mutation

Non uniform mutation changes one of the parameters of the parent according to a non-uniform probability distribution. This Gaussian distribution starts wide, and narrows to a point distribution as the current generation approaches the maximum generation.

#### d) Uniform Mutation

Uniform mutation changes one of the parameters of the parent based on a uniform probability distribution.

Crossover and mutation are used in order to increase the solution space. Besides that, these operators are also used so that a better gene can be formed to get better results for the floorplan.

The GA algorithm that is used for this project is maximization. Hence, the fitness function is negated to obtain the minimum deadspace. The deadspace of the floorplan is calculated by using the DM. DM will form the placement of the floorplan of the given blocks by referring to the randomly generated string of solutions from GA. The fitness of the random generated population will then be calculated based on the deadspace area. After the crossover and mutation functions are used, the children also need to be evaluated to determine the fitness of the functions. Only those with the fitness match to the required criterion will be kept to be brought to the next generation. The selection of the chromosome is based on the normalized geometric selection in this project.

In normal geometric selection, the population is ranked according to the fitness and also the probability of selecting the individuals which are fit instead of depending just on their fitness value. Hence, the probability for selection is given as in Figure 3.25.

$$P_i = q' \times (1 - q)^{r-1}$$

$$q' = \frac{q}{1 - (1 - q)^n}$$

where,

p = probability of selecting the individual

q = probability of selecting the best individual

r = rank of the individual (best is 1)

n = population size

Figure 3.25 Selection Probability

In this selection, the chromosomes that have higher fitness value will be ranked to the top and will have a higher probability of being selected compared to the chromosomes that have a lower fitness.

To terminate the GA algorithm, a termination criterion needs to be satisfied. GA will not obtain the optimal best solution. Hence, GA is usually terminated when it reaches the maximum number of generations. For this case, a maximum generation term is used to determine termination.

By using GA for optimization with DM as representation, we can optimize the floorplan using deadspace area as function. This algorithm is referred as DMGA (Dot Model Genetic Algorithm). The results of DMGA will be discussed in chapter 4.

### 3.2 Floorplan Optimization using Corner Bottom Left List with Genetic Algorithm

This section uses the Corner Bottom Left List to represent the floorplan and Genetic Algorithm as the optimization algorithm. Corner Bottom Left List (CBLL) is developed based on topological method. This model is a partially deterministic model where the model will place the block in a way that it has a minimum local deadspace area. After that, a modified cross entropy method is used with CBLL. CBLL uses three tuples which are represented as  $CBLL = (B_i, P_i, R_i)$ . Similar to DM,  $B_i$  also represents the block number,  $P_i$  represents the position that is placed whether is placed on the right or top of the shape of the boundary and also  $R_i$  represents the orientation of the block  $B_i$ .

Genetic Algorithm is used for this work as a global search method for optimization. GA is used to generate the representation for CBLL. GA chromosomes are modified according to the CBLL representation so that it can be used to encode CBLL to obtain optimum results.

#### 3.2.1 Corner Bottom Left List

Corner Bottom Left List (CBLL) is a floorplan representation which has three tuples,  $(B_i, P_i, R_i)$ ,  $1 \leq i \leq m$  where  $m$  represents the modules. This CBLL representation will model the floorplan according to the tuples and also gives the geometric relationship between the blocks in the floorplan. A deterministic algorithm is added in this representation to minimize the local deadspace area to reduce solution space for the heuristic optimization algorithm. This also can reduce the number of solution space that is needed for the optimization algorithm. CBLL is a compacted non-slicing floorplan representation. This means that the placement of the modules or blocks is placed in sequence in a compacted manner. CBLL uses a sequence of module names, a sequence of relationship of the next module and also a sequence of the module rotation.

### 3.2.1.1 Preliminaries

Let  $B = \{b_1, b_2, \dots, b_m\}$  be a set of  $m$  modules with the width and height denoted as  $W_i$  and  $H_i$  respectively. The area of the module  $A_i$ , can be calculated as below:

$$A_i = W_i * H_i \quad (2)$$

where,  $1 \leq i \leq m$ . Let  $(x_i, y_i)$  denote the coordinates of the bottom right of the module and  $(x_i', y_i')$  denote the coordinate of the top left of the module. A placement, namely PL, is assigned for  $(x_i, y_i)$  where no two modules are allowed to overlap for each of the  $b_i$ ,  $1 \leq i \leq m$ .

The aim of this placement is to minimize the cost metric, area. The modules are placed one at a time according to the predefined order based on the sequence of the block number. When a module is placed, a contour is formed according to the shapes of CBLL which have been fixed. The notations that are used for further description are given below.

- 1)  $B_i$  denotes the module sequence that is to be placed at a time with the left top represented by  $(x_i', y_i')$  and the bottom right represented by  $(x_i, y_i)$ .
- 2)  $P_i$  denotes the position of the module that is placed with reference to the contour. There are two positions that are used; one is on right of the contour where the block is placed in x-direction and the other is on top of the contour where the block is placed in y-direction.
- 3)  $R_i$  denotes the rotation of the module; which is no rotation or  $90^\circ$  rotation.
- 4)  $C$  denotes the contour, where  $(x^j, y^j)$  represents the corner of the contour where,  $1 \leq j \leq 4$ . A maximum of only four corners will be used for the different shapes of the boundary.
- 5)  $W_i$  denotes the width of the selected modules for placements  $(w_1, w_2, \dots, w_m)$
- 6)  $H_i$  denotes the width of the selected modules for placements  $(h_1, h_2, \dots, h_m)$



### 3.2.1.2 From CBLL to Placement

CBLL has three tuple,  $(B_i, P_i, R_i)$ ,  $1 \leq i \leq m$  where  $m$  represents the modules.  $B_i = (b_1, b_2, \dots, b_n)$  is the module sequence where  $W_i = (w_1, w_2, \dots, w_m)$  and  $H_i = (h_1, h_2, \dots, h_m)$  denote the width and height of the modules. The first block will be placed according to  $b_1$  on the plane and the rotation of the block will be referred to  $R_i$ .  $P_1$  is ignored for the first block as the first block will always be placed on the left bottom corner of the plane. Figure 3.26 shows how the first block is placed. The corner of the contour is denoted as Point 1 =  $(x^1, y^1)$  and Point 2 =  $(x^2, y^2)$ . The shape of this contour that is formed by the first block is known as rectangle. There are 12 shapes of the contour that are considered for the model of CBLL. The shapes that are used are rectangle, L-shape, stairs, N-shape, Sleep-T, T-shape, P-shape, U-stairs, B-stairs, d-shape, C-shape and U-shape. The contour shapes are shown in Figure 3.27. The corners of the contour are also shown in Figure 3.27. The maximum number of corners which is used for CBLL is 4 corners, which are Point 1 =  $(x^1, y^1)$ , Point 2 =  $(x^2, y^2)$ , Point 3 =  $(x^3, y^3)$  and Point 4 =  $(x^4, y^4)$ .

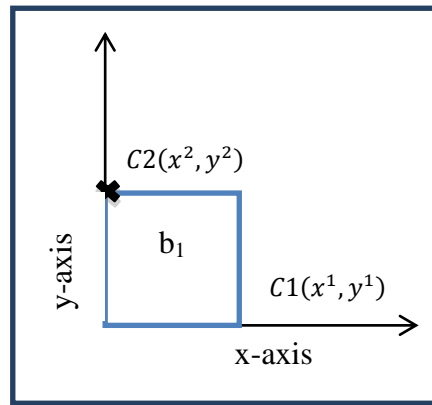
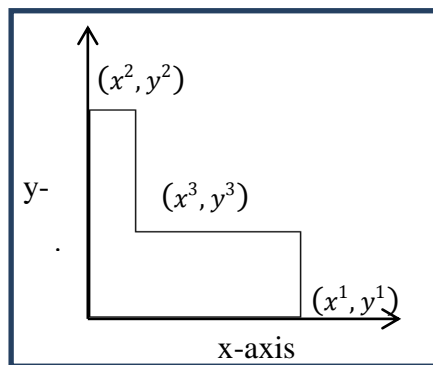
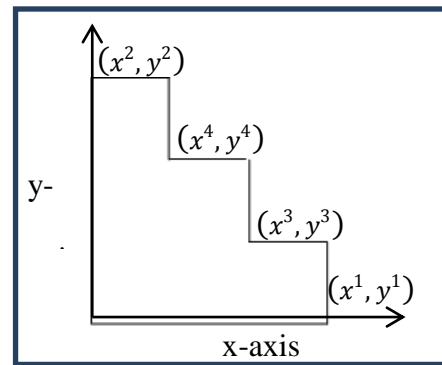


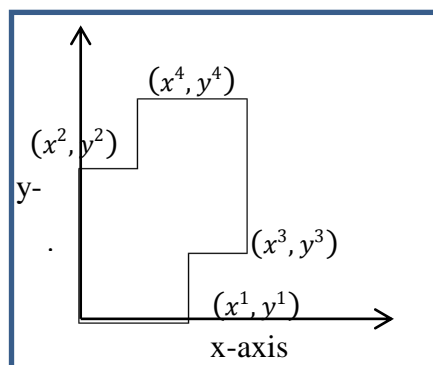
Figure 3.26 Placement of the first block with shape rectangle



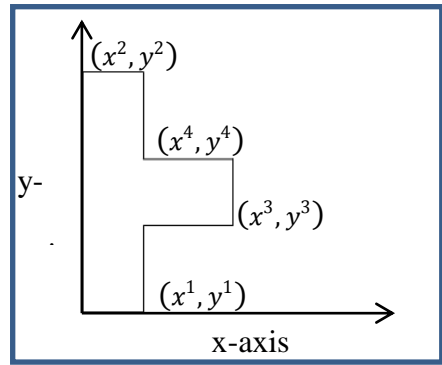
(a) L-shape



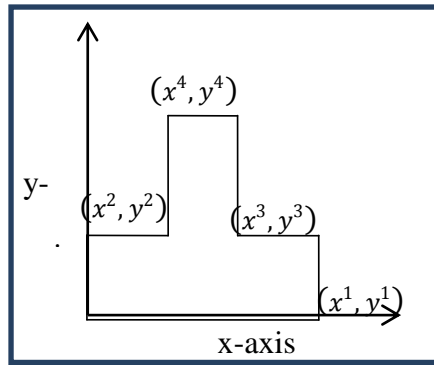
(b) stairs



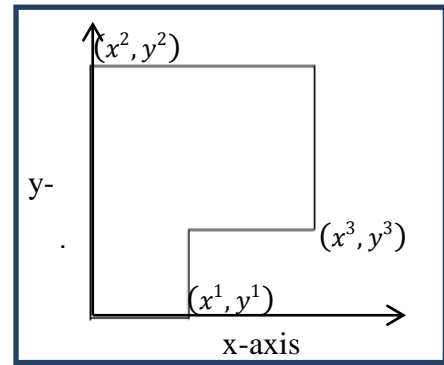
(d) N-shape



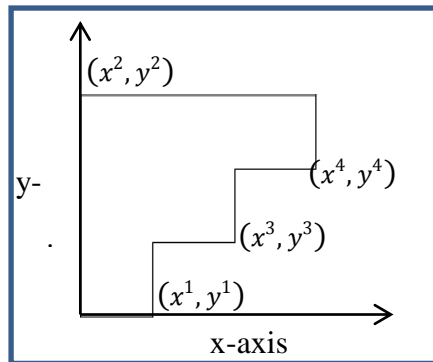
(d) sleep-T



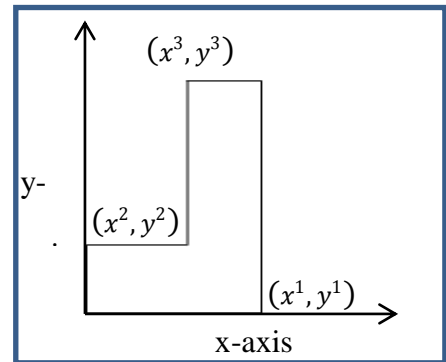
(e) T-shape



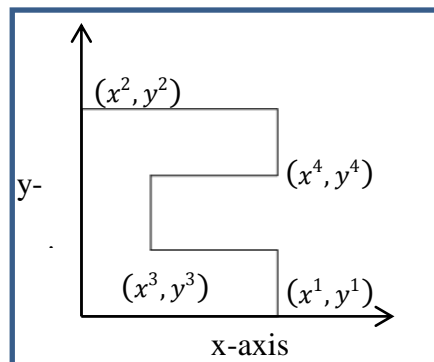
(f) P-shape



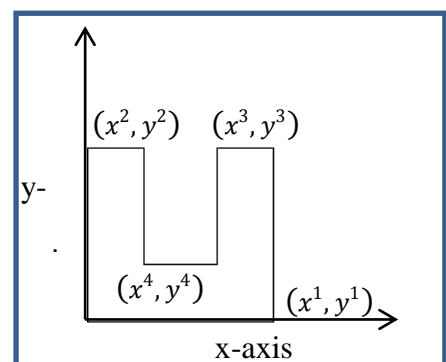
(g) U-stairs



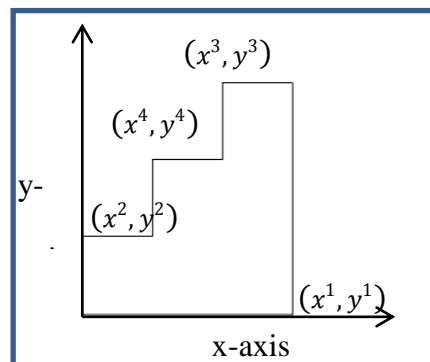
(h) d-shape



(i) C-shape



(j) U-shape



(k) b-stairs

Figure 3.27 Contour shape and their corners

In order to place the next block,  $b_2$ , we need to check the shape of the contour. The second block that is to be placed always has a previous contour of rectangle. Hence, we can place it either on the right of the contour or on top of the contour according to the position given in the representation,  $P_2$  of the CBLL. The rotation of  $b_2$  depends on  $R_2$ . After placing the module,  $b_2$ , we need to get the new contour shapes and coordinates.

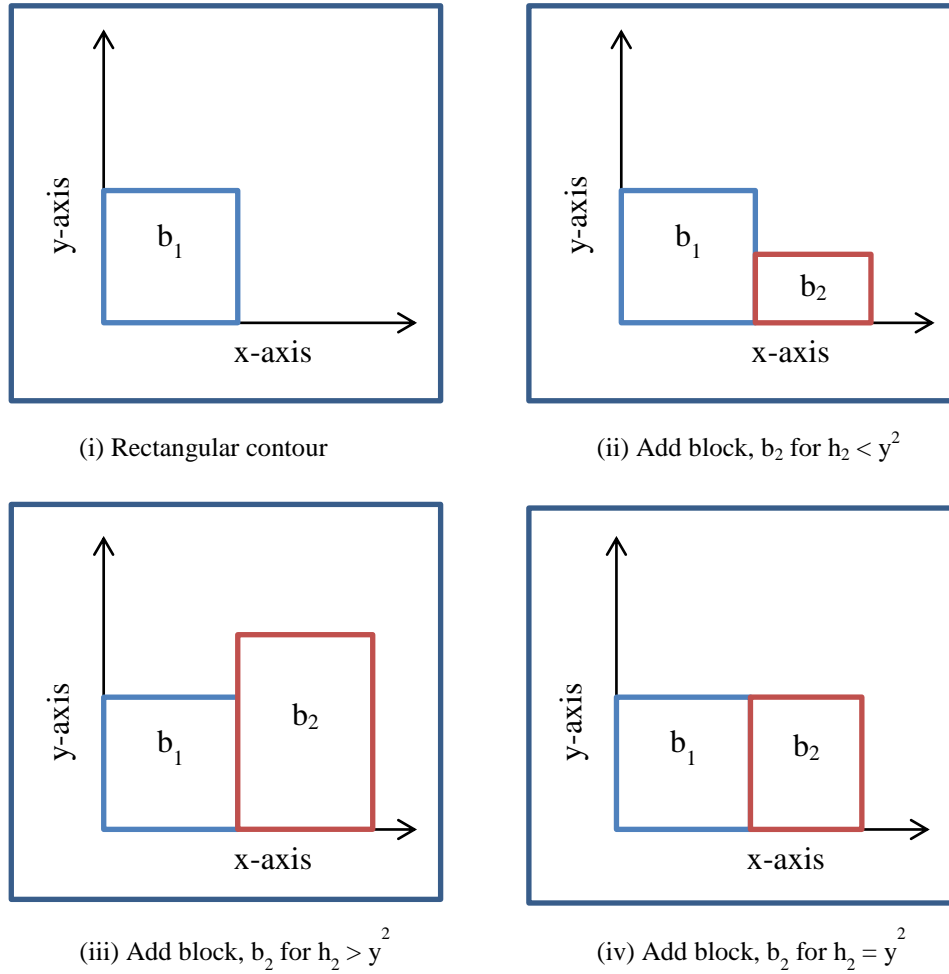


Figure 3.28 Placing of block and updating the contour

Given that the rectangle contour has the corners,  $C1 = (x^1, y^1)$  and  $C2 = (x^2, y^2)$  as shown in Figure 3.28(i). If the block,  $b_2$  is placed on the right of the contour, we will get L-shape if the current block height,  $h_2 < y^2$  as shown in Figure 3.28(ii). Hence, the contour corners will be changed to  $C1 = (x^1 + w_1, y^1)$ ,  $C2 = (x^2, y^2)$  and  $C3 = (x^1, h_1)$ . If the

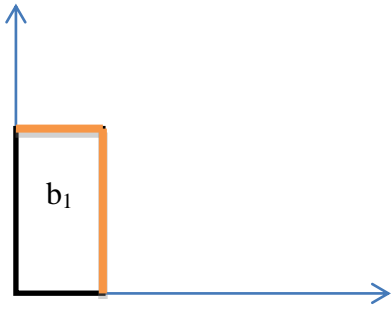
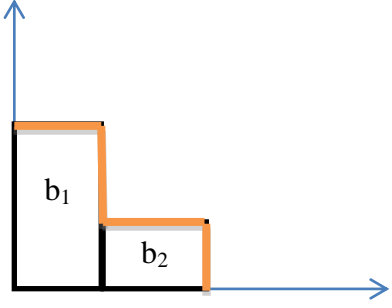
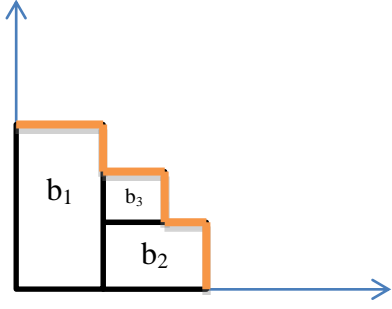
block height  $h_2 = y^2$  as shown in Figure 3.28(iii), we will get back the rectangle contour and the contour corners will become  $C1 = (x^1 + w_1, y^1)$  and  $C2 = (x^2, y^2)$ . If the current block height,  $h_2 < y^2$  as shown in Figure 3.28(iv), we will get d-shape. Hence, the contour corners will become  $C1 = (x^1 + w_1, y^1)$ ,  $C2 = (x^2, y^2)$  and  $C3 = (x^1, h_1)$ . We can observe that the contour for L-shape and also d-shape have three points. The packing of the floorplan for the above situation is shown in Figure 3.28. This packing will show how the contour shape changes depending on the block that is placed during the packing process.

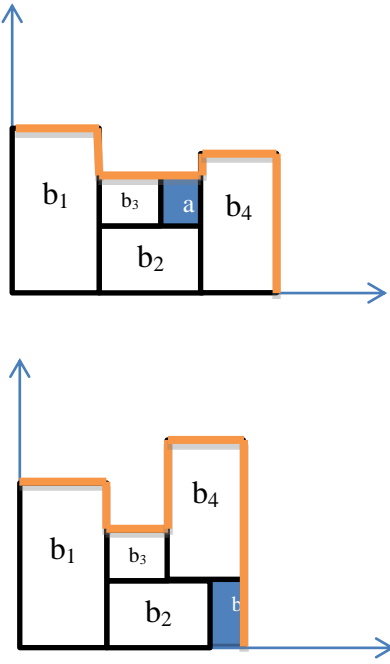
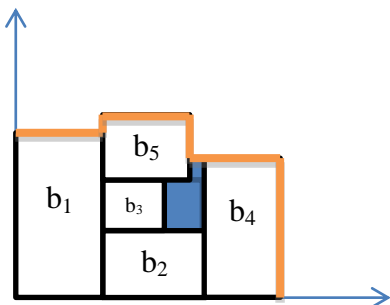
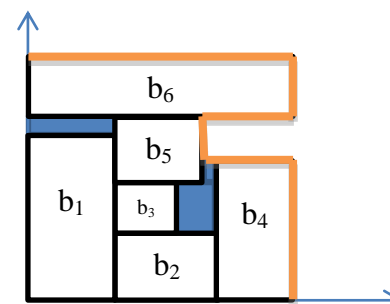
Here, we will discuss about how the placement is done together with some deterministic algorithm that calculates the minimum local deadspace area that is to be locked during placement in order for the contour to return to the 12 shapes. It is important to return the contour to one of the 12 shapes as CBLL uses these shapes to form placement. The locked areas will be taken as deadspace area as CBLL is done based on the shape of the contours. Besides that, CBLL also will move the blocks according to the size of the contour and the blocks given so that no two blocks overlaps one another. This is an important criterion for floorplanning optimization.

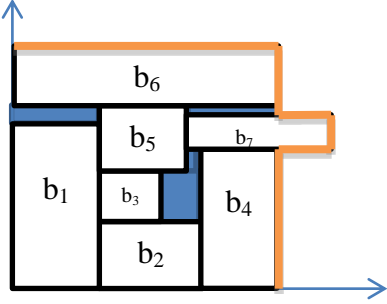
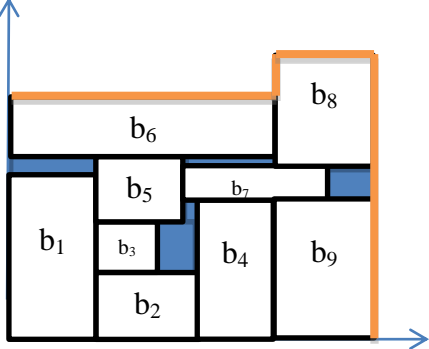
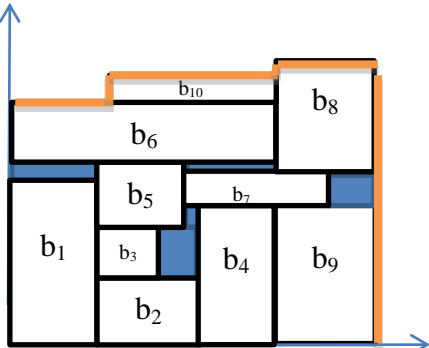
Given  $CBLL = \{(b_1b_2b_3b_4b_5b_6b_7b_8b_9b_{10}b_{11}b_{12})(000011010111)(110010110000)\}$

where the packing and placement of the modules are shown in Table 3.1. The first tuple of CBLL represents the block sequence being selected. The second tuple of CBLL represents the position that will be placed based on the contour where 0 means on the right of the contour and 1 means on the top of the contour. This position is also partially determined by the deterministic algorithm in the CBLL depending on the shapes of the contour. Table 3.1 also shows how the deterministic algorithm functions during placement. The third tuple represents the orientation of the block where 1 gives a  $90^0$  rotation to the block and 0 means no rotation is done.

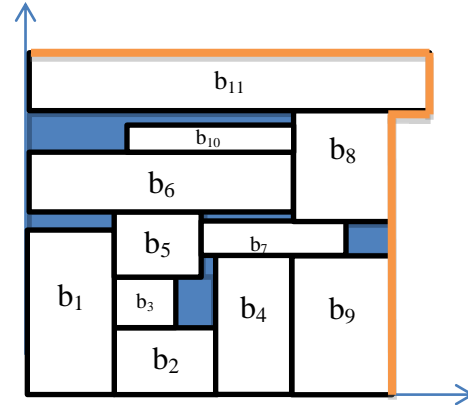
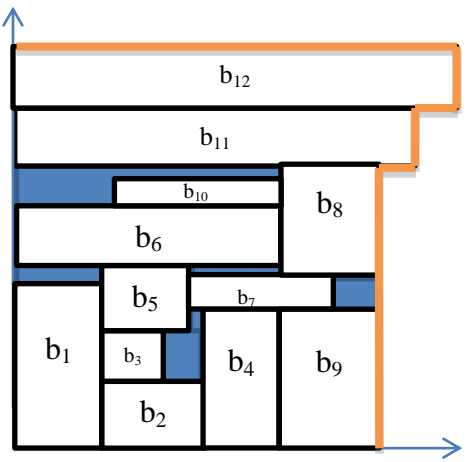
Table 1: Packing and Placement

	<p><math>CBLL_1 = (b_1, 0, 1)</math></p> <p>The plane is empty. Hence, block <math>b_1</math> is placed on the left bottom of the plane and is rotated <math>90^0</math>. The shape of the contour is rectangle. This can be seen in the figure beside.</p>
	<p><math>CBLL_2 = (b_2, 0, 1)</math></p> <p>The previous contour is rectangle. Hence, block, <math>b_2</math> is placed on the right of the rectangle contour and is rotated <math>90^0</math>. The shape of the contour becomes L-shape. This can be seen in the figure beside.</p>
	<p><math>CBLL_3 = (b_3, 0, 0)</math></p> <p>The previous contour is L-shape. Hence, block, <math>b_3</math> is placed on the L-shape without referring to the position as the focus is to pack to the left corner instead of growing the placement horizontally or vertically. This is part of the deterministic algorithm instead of depending fully on the representation. This only happens in L-shape. Hence, the contour shaped produced is stairs as shown in the figure beside.</p>

	<p><math>CBLL_4 = (b_4, 0, 0)</math></p> <p>The previous contour shape is stairs. The block, <math>b_4</math> should be placed on the right of the contour. There are 2 locations which can be placed as shown in the figure beside. Hence, the area, <math>a</math> and <math>b</math> are calculated. Area <math>a</math> and <math>b</math> are areas that are locked after the contour is formed. Hence, the placement with lower locked area will be used. In this sample, area <math>a</math> is selected. Then, the contour shape now becomes U-shape</p>
	<p><math>CBLL_5 = (b_5, 1, 1)</math></p> <p>The previous contour is U-shape. As the height of <math>b_5</math> is smaller than the width of <math>b_2</math>, <math>b_5</math> is placed inside the U-shape on the top and is rotated <math>90^\circ</math>. Then, the contour shape becomes T-shape.</p>
	<p><math>CBLL_6 = (b_6, 1, 0)</math></p> <p>The previous contour is T-shape. Block <math>b_6</math> is placed on top of the contour as shown in the figure beside to form C-shape contour.</p>

	<p><math>CBL_{L_7} = (b_7, 0, 1)</math></p> <p>The shape of the previous contour is C-shape. The module, <math>b_7</math> is rotated <math>90^0</math> and is placed beside the contour as shown in the figure beside. Then the new contour shape is sleep-T.</p>
	<p><math>CBL_{L_9} = (b_9, 0, 0)</math></p> <p>The previous contour is N-shape. The block <math>b_9</math> is placed beside the contour as shown in the figure beside and a new contour with the shape d-shape is formed.</p>
	<p><math>CBL_{L_{10}} = (b_{10}, 1, 0)</math></p> <p>The previous contour is d-shape. The block, <math>b_{10}</math> is placed on top of the contour as shown in the figure beside. Hence, the new contour has the shape of b-stairs.</p>



	<p><math>CBL_{11} = (b_{11}, 1, 0)</math></p> <p>The previous contour is b-stairs. The block <math>b_{11}</math> is placed on top of the contour as shown in the figure beside. Hence, the new contour formed has the shape of P-shape.</p>
	<p><math>CBL_{12} = (b_{12}, 1, 0)</math></p> <p>The previous contour is P-shape. The block <math>b_{12}</math> is placed on top of the contour as shown in the figure beside. Hence, the new contour formed has the shape of U-stairs.</p>

From table 1, we can conclude that CBL is a partially deterministic representation where there are certain conditions where the algorithm needs to calculate to determine which location the block is to be placed based on the shape of the contour. In order to calculate the total deadspace area,  $D$ , we get the height of the boundary,  $H$  and the width of the boundary,  $W$  to get the area of the placement minus the total area of the modules given. The percentage of the deadspace area is as the following formula:

$$D = \frac{W * H - \sum_{i=1}^m w_i * h_i}{\sum_{i=1}^m w_i * h_i} * 100\% \quad (3)$$

### 3.2.2 Implementation of GA and CBLL

This GA also uses the same source code as the previous method. From this source code, a slight modification is done in order to use for CBLL. CBLL consists of three tuples which gives  $(B_i, P_i, R_i)$ . In order to use GA, we need to model the chromosomes so that it matches both CBLL and also GA in order to do optimization. Hence, the chromosome for CBLL is modelled as in Figure 3.29.

$$\text{Chromosome} = \{(B_1, B_2, \dots, B_n) (P_1, P_2, \dots, P_n) (R_1, R_2, \dots, R_n) \}$$

Figure 3.29 Chromosome Model

B represents the block number. This number must be permuted numbers as one block can only be placed once in the floorplan. This is obtained by using random permutation in matlab function. P represents the position of the block. The position of the block can be either right or top of the block. Hence, only 0s and 1s are used where 0 represents right and 1 represents top. In order to generate these random variables, formula in Figure 3.30 is used.

$$R = \text{round}(\text{rand}(b_1, \dots, b_n))$$

Figure 3.30 Random Binary Variables

Finally, R represents the orientation of the block . This representation is also 0s and 1s. In order to generate these random variables, the formula in Figure 3.31 is used.

$$R = \text{round}(\text{rand}(b_1, \dots, b_n))$$

Figure 3.31 Random Binary Variables

After modelling the chromosomes, then we need to initialize the initial population according to the random generation as stated earlier. Same as method 1, the number of populations used is determined by the number of blocks that are needed for optimization. When the number of blocks to be placed in floorplan increases, the number of population needs to be increased.

After initializing the population, all chromosomes are evaluated using CBLL in order to obtain the fitness of the chromosomes. Then the population is arranged according to the fitness of the chromosomes where the fittest chromosome will be on the top and the worst chromosome will be at the bottom. The fitness of the chromosomes depends on the deadspace area of the floorplan. The lower the deadspace area, the fitter the chromosome and vice versa. CBLL is used to calculate the deadspace area of the floorplan of the chromosomes or also are known as the solution strings. After arranging the population, the top quantile is selected and will be brought to the next population. In this new population, a few chromosomes are selected to do the GA operators such as Crossover and also Mutation.

There are many types of crossover operations. But the crossover operators that are used in this work are simple crossover for the binary representation. The crossovers that are used for the ordered based representations are the cyclic crossover, order-based crossover, single point crossover and partial mapping crossover. The crossover methods were described in the first section.

Besides crossover operator, a few of the chromosomes from the population is selected to do mutation operations. There are 5 types of mutation operator for order based part of the chromosomes which are used in this section. They are inversion mutation, swap mutation, adjacent swap mutation, three swap mutation and shift mutation. Besides mutation for order-based number, mutation also is done for the binary representation. There are two types of mutation operations which are used in this work which are the inversion mutation and binary mutation. Crossover and mutation are used so that more solutions can be produced. Besides that, these can give better gene to get better results for floorplan.

The GA algorithm that is used for this project is for maximization same as in the previous section. Hence, negative is added to the deadspace area to get the fitness of the solution string to minimize the deadspace. In this section, the deadspace will be calculated using CBLL. CBLL will also give the placement of the blocks according to the solution strings generated by GA. Hence, the fitness of the solution string can be determined. After doing some operator functions, the fitness of the children needs to be evaluated before inserting them in to the population again. Only those with the fitness match to the required criterion will be kept to be brought into the next generation. The selection of the chromosome is based on the normalized geometric selection in this project as in the first method.

To terminate the GA algorithm, a termination criterion needs to be met. Since GA is an optimization algorithm, it will not be able to obtain the ideal solution. Hence, GA is usually terminated when it reaches the maximum number of generation. For this case, a maximum generation term function is used to determine when to end the GA algorithm. Usually the GA algorithm is terminated by selecting the number of generations of GA.

Hence, by using GA for optimization with the use of CBLL as representation, we can optimize the floorplan based on the area of the deadspace. It will be referred as CBLL-GA (Corner Bottom Left List Genetic Algorithm) in the rest of the thesis. The results for CBLL-GA will be shown in chapter 4.

### **3.3 Floorplan Optimization using Corner Bottom Left List with modified Cross Entropy Method**

In this section, the method that is used for representation is Corner Bottom Left List (CBLL) which is developed based on topological method as discussed previously.

A modified cross entropy method is used to for optimization with CBLL as the representation. CBLL uses three tuples which are represented as  $CBLL = (B_i, P_i, R_i)$  where  $B_i$  represents the block number,  $P_i$  represents the position that is placed whether is placed on the right or top of the shape of the boundary and also  $R_i$  represents the orientation of the block  $B_i$ . The CBLL method will not be discussed in this section as it is already discussed in 3.2.1.

The CE method is modified to suit the representation of CBLL. The CE equation that has been modified to 3 dimension to form 2 sequences reflects the relationship between the blocks and the probability of the most matching pair of the CBLL representation and reduces both the local deadspace area and also the global deadspace area by using the optimal value,  $\gamma$ . CE is extended into 3 dimensional matrixes instead of the original TSP method which uses 2 matrixes for optimization. Hence, the CE method can be fully utilized based on the CBLL representation. The solution string for CBLL representation is also modified to accommodate both CE and also for the probability matrix.

### **3.3.1 Modified Cross Entropy Method**

In this section, the modified CE method will be discussed and also how the CBLL is implemented on the modified CE method. The original CE only produces a single string of permuted numbers. However, to generate the CBLL representation and also to optimize the floorplan, modifications need to be done to suit this application. The CE method is adapted from the TSP method. However, instead of maintaining the form, we change the original CE two dimension probability transition matrixes into three dimensions matrixes. The basic concept of CE is still retained as it still consists of two iterative phases which are:

- 1) Generating a random data according to the CBL representation for this project.

- 2) Updating the parameters of the random mechanism which is the parameters of the pdfs (probability density function) on the data based on the sample results so that better samples can be produced in the next iteration and the iteration stops when the samples reaches the required criteria.

### 3.3.1.1 Cross Entropy Method

The method CE employed to do optimization is as follows. As floorplanning is a minimization problem, hence we need to define a function,  $S(x)$  that needs to be minimized for some set  $X$ . The minimum is denoted with optimal value  $\gamma^*$ , as shown in the equation below:

$$\gamma^* = \min_{x \in X} S(x) \quad (4)$$

To optimize the floorplan through the CBLL representation, the minimum of the cost function  $S(x)$  is given by:

$$\min S(x) = \min\{(h_b * w_b) - \sum_{i=1}^n (h_i * w_i)\} \quad (5)$$

where  $h_b$  is the height of the floorplan boundary,  $w_b$  is the width of the floorplan boundary,  $h_i$  is the height of the module and  $w_i$  is the width of the module where  $1 \leq i \leq m$ . Next, the deterministic problem is randomized by defining a family of pdfs  $\{f(\cdot; \mathbf{v}), \mathbf{v} \in V\}$  for the set of  $X$ . The estimation value  $\ell$  is a value estimated by the CE method so that floorplanning optimization can be done. The estimation value is determined as below:

$$\ell = \mathbb{P}(S(X) \geq \gamma) = \mathbb{E}_u I_{\{S(X) \geq \gamma\}} \quad (5)$$

where  $\mathbb{P}$  is the probability measure under which the random state  $X$  has pdf  $f(\cdot; \mathbf{v})$  where  $\mathbf{X}$  represents the solution strings that will be formed to get the CBLL sequence and  $\mathbb{E}_u$  denotes the corresponding expectation operation.

This estimation problem will be called as associated stochastic problem. The random vector obtained is required to be converted to CBL form. Hence, estimation  $\ell$  and the root of the equation  $\gamma$ , can be calculated.  $S(\mathbf{X}) \geq \gamma$  is a rare event and the estimation of  $\ell$  is nontrivial. The CE formulas shown in equations 6,7,8,9 and 10 are used to solve the floorplanning problem effectively by making adaptive changes to the probability density function according to the Kullback-Leibler cross entropy. This will create a sequence of  $f(\cdot; \mathbf{u})$ ,  $f(\cdot; \mathbf{v}_1)$ ,  $f(\cdot; \mathbf{v}_2), \dots$  pdfs in order to achieve the optimal floorplan area. Let  $\varrho$  be a value lying between  $10^{-2}$  and  $10^{-1}$ . Below is the fundamental of cross entropy method.

1. Adaptive updating of  $\gamma_t$  where  $t$  represents the iteration number. For a fixed  $\mathbf{v}_{t-1}$ ,

let  $\gamma_t$  be a  $(1 - \varrho)$ -quantile of  $S(\mathbf{X})$  under  $\mathbf{v}_{t-1}$ , so that  $\gamma_t$  satisfies

$$\mathbb{P}_{\mathbf{v}_{t-1}}(S(\mathbf{X}) \leq \gamma_t) \geq \varrho \quad (7)$$

$$\mathbb{P}_{\mathbf{v}_{t-1}}(S(\mathbf{X}) \geq \gamma_t) \leq \varrho \quad (8)$$

where  $\mathbf{X} \sim f(\cdot; \mathbf{v}_{t-1})$ . A simple estimator  $\hat{\gamma}_t$  of  $\gamma_t$  is the order statistic

$$\hat{\gamma}_t = S_{([ (1-\varrho)N ])} \quad (9)$$

where  $N$  is the total number of samples in the data.

2. Adaptive updating of  $\mathbf{v}_t$ . For a fixed  $\gamma_t$  and  $\mathbf{v}_{t-1}$  derive  $\mathbf{v}_t$  from the solution of the CE program:

$$\min_{\mathbf{v}} D(\mathbf{v}) = \min_{\mathbf{v}} \mathbb{E}_{\mathbf{v}_{t-1}} I_{\{S(\mathbf{X}) \leq \gamma_t\}} W(\mathbf{x}; \mathbf{u}, \mathbf{v}_{t-1}) \ln f(\mathbf{X}; \mathbf{v}) \quad (10)$$

The stochastic counterpart of (6) is as follows: for the fixed  $\hat{\gamma}_t$  and  $\hat{\mathbf{v}}_{t-1}$  derives  $\hat{\mathbf{v}}_t$  for the following program:

$$\min_{\mathbf{v}} \hat{D}(\mathbf{v}) = \min_{\mathbf{v}} \mathbb{E}_{\mathbf{v}_{t-1}} I_{\{S(\mathbf{X}) \leq \gamma_t\}} W(\mathbf{x}; \mathbf{u}, \hat{\mathbf{v}}_{t-1}) \ln f(\mathbf{X}; \mathbf{v}) \quad (11)$$

**Proposition 1:** Let  $\gamma^*$  be a minimum value for the deadspace area according to the set of  $\mathbf{X}$ . Suppose that the corresponding minimizer is unique  $\mathbf{x}^*$  and that the class

densities  $\{f(\cdot; \mathbf{v})\}$  used in the CE program contains the degenerate density with mass at  $\mathbf{x}^*$ :

$$\delta_{x^*} = \begin{cases} 1, & \text{if } x = x^* \\ 0, & \text{otherwise} \end{cases} \quad (12)$$

Hence, the solutions for the random generation and CE program in estimating  $\mathbb{P}(S(\mathbf{X}) \leq \gamma^*)$  coincide and will respond to  $\delta_{x^*}$ . From this, better solutions for  $\mathbf{X}$  can be obtained in the next iteration. This enables an estimation of the true optimal solution which is the minimum deadspace area of a floorplan as it can converge to a smaller neighbourhood for the next iteration. Instead of updating the parameter vector  $\mathbf{v}$ , directly via the (10), the following smoothed version is used:

$$\hat{\mathbf{v}}_t = \alpha \hat{\mathbf{v}}_t + (1 - \alpha) \hat{\mathbf{v}}_{t-1} \quad (13)$$

where  $\hat{\mathbf{v}}_t$  is the parameter vector obtained from (10) and  $\alpha$  is the smoothing parameter with  $0.7 < \alpha < 1$ . When  $\alpha = 1$ , the original updating rule is maintained. Smoothing is done to values of  $\hat{\mathbf{v}}_t$  in order to prevent the probability that some component  $\hat{v}_{t,i}$  of  $\hat{\mathbf{v}}_t$  will be 0s or 1s in the first few iterations. This is important when  $\hat{\mathbf{v}}_t$  is a vector or a matrix of probabilities. Note that for  $0 < \alpha < 1$ ,  $\hat{v}_{t,i} > 0$  while for  $\alpha = 1$  the first few iterations will cause  $\hat{v}_{t,i} = 0$  or  $\hat{v}_{t,i} = 1$  for some indices  $i$  and this might cause convergence to undesired solutions. (Rubinstein & Kroese, 2008)

### 3.3.1.2 Random Generation of CBLL

CBLL consists of three tuples,  $(\mathbf{B}_i, \mathbf{P}_i, \mathbf{R}_i)$ ,  $1 \leq i \leq m$  where  $m$  represents the total number of modules. The probability transition matrix  $\mathbf{p}$  is a three dimensions matrices that randomly generate the representation for the CBLL, which is  $\mathbf{X}$ . The component of probability  $\hat{p}$  for the stochastic case is as follows:

$$\hat{p}_{t,(i,j,z)} = \frac{\sum_{k=1}^N I_{\{S(\mathbf{X}) \leq \hat{\mathbf{Y}}_t\}} I_{\{\mathbf{X}_k \in \mathcal{X}_{ijz}\}}}{\sum_{k=1}^N I_{\{S(\mathbf{X}) \leq \hat{\mathbf{Y}}_t\}}} \quad (14)$$



This means that  $p_{(i,j,z)}$  corresponds to the probability of the block  $i$ , being chosen at  $j$ -th and  $z$ -th place.  $i$  represents the block number and  $j$  and  $z$  represent the sequence of the module  $i$ , being placed. The algorithm generation of random representation for CBLL is shown in the following:

```

1: Let  $t = 1$ ,  $b = 0$ , for all  $j \neq 1$ ,  $i = 1$ 
2: Generate  $U \sim U(0, 1)$ , and
   let  $R = U * \sum_{j=1}^n (1 - b_j) p_{ijz} + R$ 
3: Let sum = 0 and  $j = 0$ 
4: while sum <  $R$  do
5:    $j = j + 1$ 
6:   if  $j > n$ 
7:      $z = z + 1$ 
8:      $j = 1$ 
9:   end
10:  if  $b_j = 0$ 
11:    sum = sum +  $p_{ijz}$ 
12:  end
13: end
14: set row  $P(:,j) = 0$ 
15: normalize the row  $P_j$  to sum up to 1
16: Set  $t = t + 1$ ,  $X_t = j$ ,  $b_j = 1$  and  $i = j$ 
17: if  $t = n$ 
18:   stop
19: else return to 2
20: end

```

A generated random permutation value  $\mathbf{x} = (x_{1k}, x_{2k}, \dots, x_{nk})$  corresponds to a unique block sequence and placement accordingly. This means that the selection of the module is done as the sequence  $x_{1k} \rightarrow x_{2k} \rightarrow \dots \rightarrow x_{nk}$  where  $1 \leq k \leq 4$ .

k	P	R
1	0	0
2	0	1
3	1	0
4	1	1

Let  $\chi$  be the set of possible placement for the floorplan where the deadspace area is calculated through CBLL calculations. The goal of floorplanning is to minimize  $S$  over the set of  $\chi$  using the CE method. In order to minimize  $S$ , a specific mechanism is needed to generate the random representation for CBLL.

To generate a random CBLL representation  $\mathbf{x} = (x_{1k}, x_{2k}, \dots, x_{nk})$ , a transition probability matrix  $p$  is used in an algorithm of trajectory generation using node placement. This algorithm is crucial to generate the CBLL representation for the

optimization of CE to generate a stochastic process  $\{X_{1k}, X_{2k}, \dots, X_{nk}\}$  according to the conditional distribution of Markov chain where each module can only be chosen once and placed at one location and orientation. The transition matrix for the stochastic case is shown in (14) and the transition matrix for the deterministic component is shown below:

$$P_{t,ij} = \frac{\mathbb{E}_{P_{t-1}} I_{\{S(X) \leq \gamma_t\}} I_{X \in \mathcal{X}_{ijz}}}{\mathbb{E}_{P_{t-1}} I_{\{S(X) \leq \gamma_t\}}} \quad (15)$$

where  $\mathcal{X}_{ijz}$  denotes the set of tours from the placement of block  $i$  being chosen at position  $j$  and orientation  $z$  of the block being placed relative to the contour.

According to the trajectory generation using node placement algorithm, consider the corresponding optimal degenerate transition matrix,  $p^*$ . If  $\gamma^*$  is the optimal deadspace area, the corresponding sequence is  $x^*$ . For any  $p_{t-1}$  solution for CE, the optimal degenerate transition matrix  $p^* = (p_{ijz}^*)$  is given by the following equation:

$$p_{ijz}^* = \begin{cases} 1 & \text{if } x^* \in \mathcal{X}_{ijz}, \\ 0 & \text{otherwise.} \end{cases} \quad (16)$$

Example of a CBL sequence is as follows:

$$b_{1,2} \rightarrow b_{2,2} \rightarrow b_{3,1} \rightarrow b_{4,1} \rightarrow b_{5,4} \rightarrow b_{6,3} \rightarrow b_{7,2} \rightarrow b_{8,4} \rightarrow b_{9,1} \rightarrow b_{10,3} \rightarrow b_{11,3} \rightarrow b_{12,3}$$

The corresponding  $P^*$  is as follow:

$$P_1^* = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

$$P_2^* = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

$$P_3^* = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

$$P_4^* = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

### 3.3.1.3 Implementing Cross Entropy Algorithm

The CE method uses the probability from the trajectory generation using node placements in order to optimize the floorplan according to CBLL. Below is the CE algorithm that is used for optimizing the floorplanning:

- 1) Choose an initial reference transition matrix  $\hat{p}_0$ , where all the off-diagonal elements in the matrix will give a sum equal to  $1/n$  where  $n$  is the number of modules. Set  $t = 1$  where  $t$  represents the number of iteration in the CE algorithm.

- 2) A random permutation  $\mathbf{x} = (x_{1k}, x_{2k}, \dots, x_{nk})$  is generated according to the trajectory generation using node placement and converted to CBLLE representation to evaluate the deadspace area in order to obtain the requisite quality of the randomly permuted sequence.  $p = \hat{p}_{t-1}$  is the probability that is generated and is computed according to the sample  $(1 - \varrho)$ -quantile  $\hat{Y}_t$ , of the performance,

$$\hat{Y}_t = S_{((1-\varrho)N)} \quad (17)$$

- 3) After updating the transition matrix  $\hat{p}_t$ , the matrix is smoothed out based on the equation below:

$$\hat{p}_t = \alpha \hat{p}_t + (1 - \alpha) \hat{p}_{t-1} \quad (18)$$

where  $\alpha$  is chosen based on the percentage of  $\hat{p}_t$ .  $\hat{p}_t$  is calculated and retained as  $\hat{p}_{t-1}$  in the next iteration.

- 4) If for some  $t = t+1$ , say  $d = 5$ ,

$$\hat{Y}_t = \hat{Y}_{t-1} = \dots = \hat{Y}_{t-d} \quad (19)$$

then the algorithm will terminate. If not, set  $t = t+1$  and reiterate from Step 2.

Next, the best  $(1 - \varrho)$  -quantile will be brought forward by increasing the probability of the node in the specific sequence of the results obtained. When it reaches the stopping criterion as described in step 4, the algorithm should converge and gives the minimum deadspace area of the given block for a floorplan. This study will be referred as CBLLE-CE. The results will be discussed in chapter 4.

## CHAPTER 4. RESULTS, DATA ANALYSIS AND DISCUSSION

### 4.1 DMGA

In the first part of this section, the effects of mutation operators and crossover operators are discussed for both floating point representation and also ordered base sequence. This is to select the optimal frequency for mutation and crossover operators. The final section discusses on the optimal results obtained using DMGA.

#### 4.1.1 Effects of mutation operators on floating point representation

In this section, the effects of mutation operations frequency for floating point representation is studied using population size of 100 for 100 generations. Benchmark hp is used for this purpose. The floating point representation consists of 2 strings which are relative block number and position of the reference block that is used in DM. The frequencies of mutation operations are varied from low value (5) to high value (30) for each mutation operator. The mutation operators that were used are boundary mutation, multi non-uniform mutation, non-uniform mutation and uniform mutation. The mutation operation frequency for each operator is varied according to the sequence mentioned. Table 2 and Figure 4.1 show the optimization results for deadspace area and time when the frequency of mutation operations are varied for floating point representation.

Table 2: Study on the Frequency of Mutation Operators

Mutations Frequency				Deadspace(%)	Time (s)
Boundary	Multi Non-Uniform	Non-uniform	Uniform		
5	5	5	5	12.92	383.25
30	5	5	5	8.79	475.24
5	30	5	5	11.41	473.25
5	5	30	5	11.77	385.04
5	5	5	30	9.47	683.64
30	30	30	30	9.71	1193.52

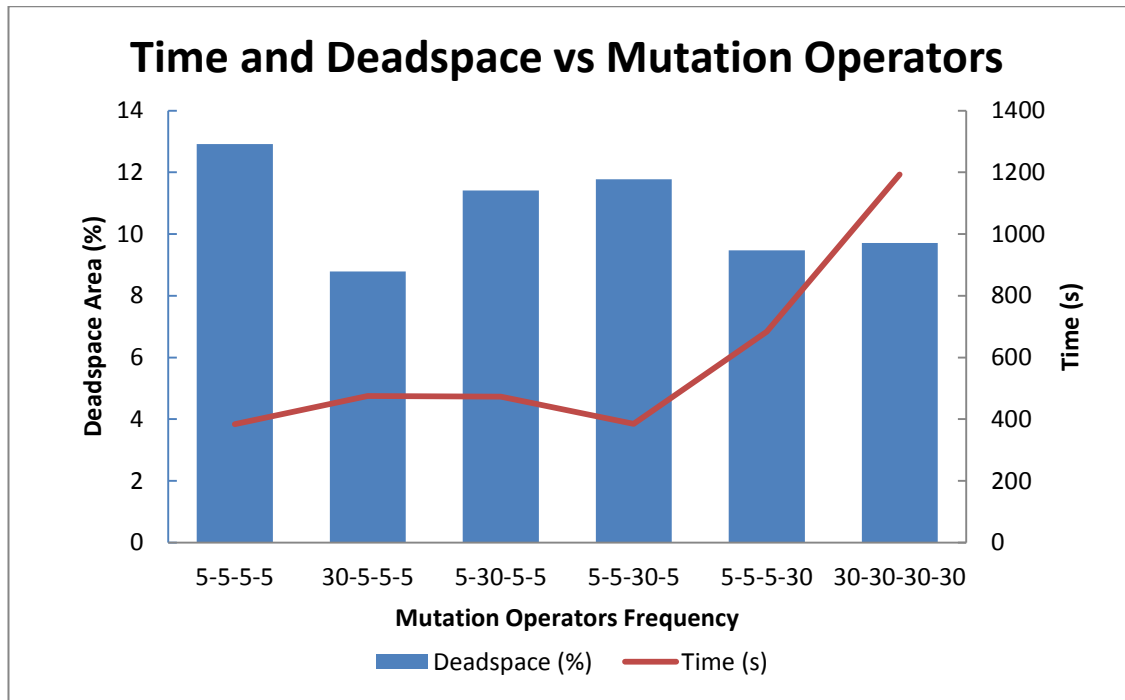


Figure 4.1 Study on Number of Mutation Operators for floating point representation

The result obtained is based on average results of 30 simulations for each case. From the results, we can observe that when all types of mutation operations frequencies are low (5), the deadspace area obtained is higher. The time consumed to complete each optimization process is the shortest compared to the others.

When boundary mutation frequency is high (30) and the rest are retained low (5), the deadspace obtained is the lowest in comparison with others but the time taken is longer compared to running the optimization when all mutation operations frequencies are retained low (5). However, when the mutation operation frequency for multi non-uniform mutation or non-uniform mutation is high (30), only marginal reduction in deadspace area and a slight increase in time for each optimization are observed. When the uniform mutation operation frequency is high (30), the deadspace area obtained is low compared to when all the mutation operations frequency are low. However, it is observed that an increase of 78% in optimization time compared to when all types of mutation operations frequency is low (5). When all the mutation operations frequencies are high, the deadspace area is also low though it is not as low as when only boundary

mutation operation frequency is allowed to be high (30). However, the time taken is considerably long. Hence, in order to select mutation operations frequencies for optimum outcome, we need to consider the time and also deadspace area results. The runtime taken and the deadspace area obtained must be as minimal as possible for efficient floorplanning optimization. Hence, the optimum frequencies of mutation operators selected are as follow:

- a. Boundary mutation – 30
- b. Multi Non-uniform Mutation – 5
- c. Non-uniform Mutation – 5
- d. Uniform Mutation – 5

#### **4.1.2 Effects of crossover operators for floating point representation**

In this section, the effect of crossover operations frequencies for floating point representation is studied with a population size of 100 for 100 generations. The benchmark used for this study is hp. The floating point representation consists of 2 different strings which are relative block number string and position of the reference block string that is used in DM. The crossover operations frequencies are varied from low value (5) to high value (30) for each crossover operator. The crossover operators that were used are arithmetic crossover, heuristic crossover and simple crossover. The crossover operations frequencies are varied according to the sequence mentioned. Table 3 and Figure 4.2 show the optimization results for deadspace area and time when the frequency of crossover operations are varied for floating point representation.

The result obtained is based on the average results of 30 simulations for each case study. From the results, we can observe that when all crossover operations frequencies are low (5) for each crossover operator, the deadspace area obtained is the highest in comparison with other crossover operations frequencies combination.

However, the time taken to complete each optimization is the shortest compared to the others.

Table 3: Study on the Frequency of Crossover

Crossover Frequency			Deadspace(%)	Time (s)
Arithmetic	Heuristic	Simple		
5	5	5	12.92	383.25
30	5	5	12.04	503.44
5	30	5	10.28	432.94
5	5	30	12.08	429.75
30	30	30	12.26	874.51

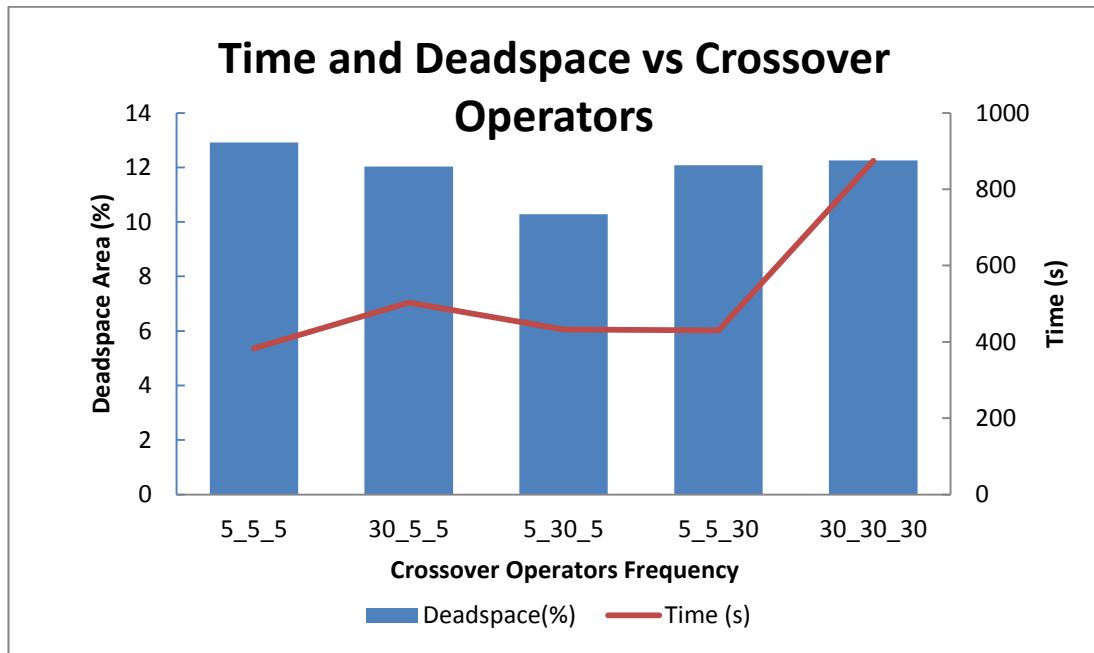


Figure 4.2 Study on the Number of Crossover for Floating Point Representation

When the simple crossover operator is high (30) and the other crossover operators are retained low, the deadspace obtained is the lowest compared. However, the time taken is 12% longer that when all the frequencies of crossover operations are low. There is no significant difference in the deadspace area when the frequency of crossover operator for arithmetic crossover or heuristic crossover or when all crossover operations frequencies are high (30). The runtime for high frequency of arithmetic



crossover or heuristic crossover are slightly longer compared to when all the crossover operations frequencies are low (5). However, when all the crossover operations frequencies are high (30), the time taken to complete an optimization is considerably long. Hence, similar to selection of mutation operations frequencies, selecting the crossover operations frequencies also involves both time and deadspace area. Hence, the optimum frequencies of crossover operators are as follow:

- a. Arithmetic Crossover – 5
- b. Heuristic Crossover – 30
- c. Simple Crossover – 5

#### **4.1.3 Effects of mutation operators for ordered based sequence**

In this section, the effects of mutation operators for ordered based sequence is studied with a population size of 100 for 100 generations. The benchmark used for this study is hp. The ordered based sequence used is for the current block placement that is used in DM. Hence this will affect the sequence where the blocks are chosen. The effects of the mutation operators which are studied are in the sequence of inversion mutation, adjacent swap mutation, shift mutation, swap mutation and threeswap mutation. The mutation operations frequencies are varied from low value (5) to high value (30). Table 4 and Figure 4.3 show the optimization results for deadspace area and time when the frequency of mutation operators are varied for ordered based sequence.

Table 4: Study on the Frequency of Mutation

Mutations Frequency					Deadspace (%)	Time (s)
Inversion	Adjacenet Swap	Shift	Swap	Threeswap		
5	5	5	5	5	12.92	383.25
30	5	5	5	5	10.74	473.53
5	30	5	5	5	12.44	497.63
5	5	30	5	5	10.12	435.98
5	5	5	30	5	12.74	478.83
5	5	5	5	30	11.69	489.54
30	30	30	30	30	14.26	987.43

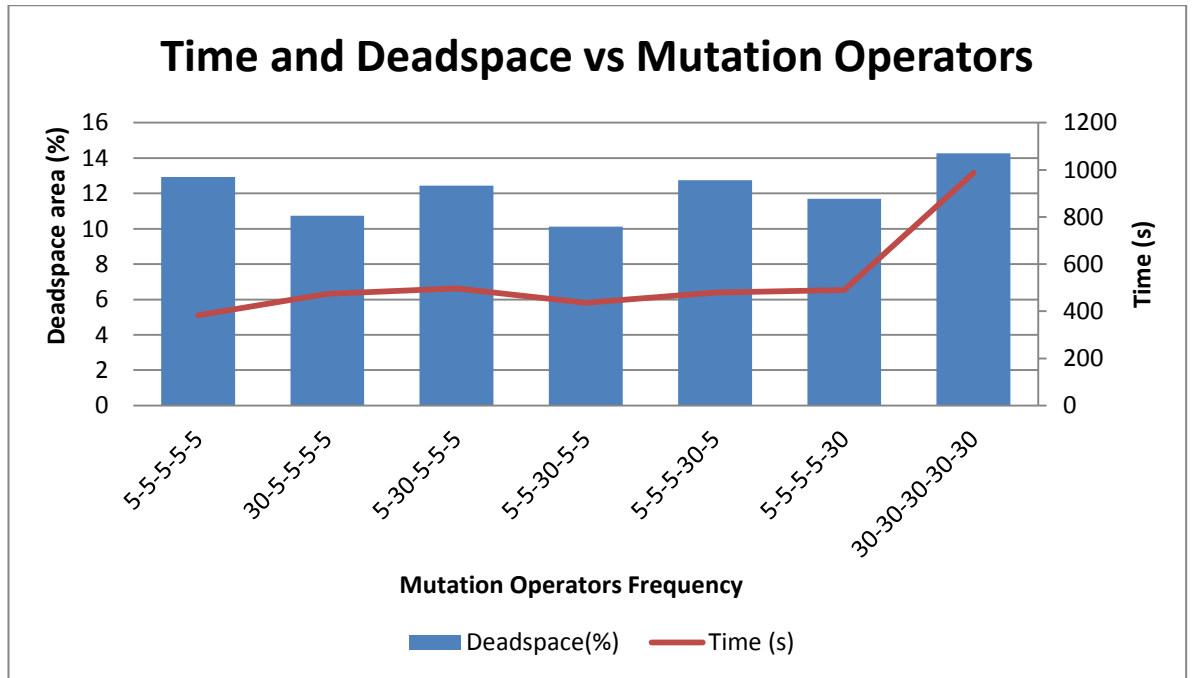


Figure 4.3 Study on the Frequency of Mutation for ordered based number

The result obtained is based on average results of 30 simulations for each case study. From the results, we can observe that when all the crossover operations frequencies are high (30), the deadspace is highest in comparison with other mutation parameter combinations and the time taken to complete the optimization is considerably long. When the inversion mutation frequency or the shift mutation operator frequency is high (30), the deadspace area is the lowest compared to frequency combinations.

However, the time taken is 15% longer compared to when all the mutation operations frequency is retained low. There is only marginal decrease in deadspace area when the mutation operations frequency for adjacent swap mutation, swap mutation or threeswap mutation is high (30). It can be concluded that operator frequency for inversion mutation and shift mutation should be high (30) as this will reduce the deadspace area and while causing minimal increase in the time taken to complete the simulation. Having all high (30) for the different types of mutation operations frequencies is definitely a bad choice as it does not only worsen the deadspace area but also uses more time to complete a simulation. Hence, the optimal frequency of crossover operator that is selected is as follow:

- a. Inversion Mutation – 5
- b. Adjacent Swap Mutation – 5
- c. Shift Mutation – 30
- d. Swap Mutation – 5
- e. Threeswap Mutation – 5

#### **4.1.4 Effects of crossover operators for ordered based numbers number**

In this section, the effects of crossover operators for ordered based numbers is studied with population size of 100 for 100 generations. The benchmark used for this study is hp. Ordered based sequences are used for the current block placement. Hence, this will affect the sequence where the blocks are chosen. The effects of the crossover operators that are studied are cyclic crossover, single point crossover, order-based crossover, uniform crossover and partial mapping crossover. The frequency of crossover operations is varied from low value (5) to high value (30). Table 5 and Figure 4.4 show the effects in the optimization results for deadspace area and time when the frequency of crossover operators are varied for ordered based sequence.

Table 5: Study on the Frequency of Crossover

Crossover Frequency					Deadspace (%)	Time (s)
Cyclic	Single Point	Order-based	Uniform	Partial Mapping		
5	5	5	5	5	12.92	383.25
30	5	5	5	5	11.23	368.44
5	30	5	5	5	12.61	365.90
5	5	30	5	5	10.64	367.29
5	5	5	30	5	10.89	367.08
5	5	5	5	30	11.46	376.09
30	30	30	30	30	7.88	423.84

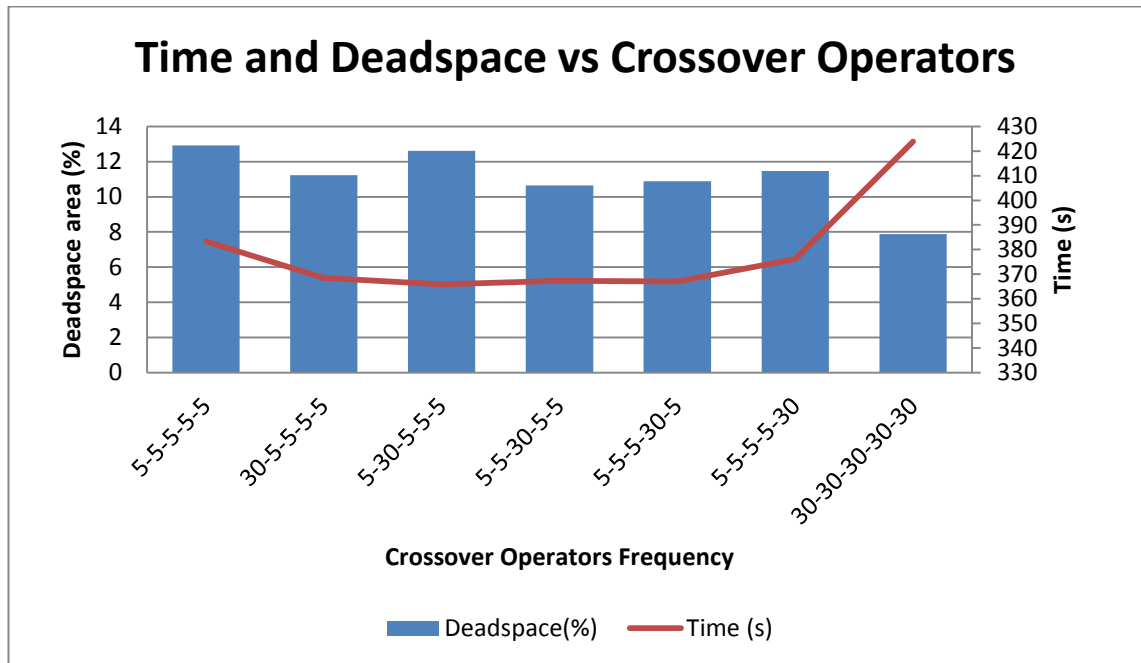


Figure 4.4 Study on the Frequency of Crossover for ordered based number

The result obtained is based on average results of 30 simulations for each case. From the results, we can observe that when all the crossover operators frequencies are high (30), we obtain the lowest deadspace area. The time taken is only slightly longer compared to when all the crossover operators frequency are low (5).

Introducing single point crossover frequency at 30 hardly brings any changes to the deadspace area though the time taken is marginally smaller in comparison with the

simulation that involves all crossover operations frequencies low. When the frequency for cyclic crossover, order-based crossover, uniform crossover or partial mapping crossover is high (30), the deadspace area is relatively lower compared to when all are low (5) but higher compared to when all are high (30). Hence, when all the crossover operator frequencies are high will give a better result of deadspace area and does not affect the time taken to complete the crossover. Hence, the optimal frequency of crossover operators that is selected is as follow:

- a. Cyclic Crossover – 30
- b. Single Point Crossover – 30
- c. Order-based Crossover – 30
- d. Uniform Crossover – 30
- e. Partial Mapping Crossover - 30

#### 4.1.5 Optimal Results and Data Analysis

For DMGA, a few analyses had been done in order to test validity of the work. Table 6 and Figure 4.5 summarises the results of the benchmark for apte, hp, xerox, ami33 and ami49.

Table 6: Optimal Results for DMGA

Benchmark	Apte	xerox	hp	ami33	ami49
Number of blocks, n	9	10	11	33	49
Minimum deadspace, (%)	1.5887	5.2593	6.7784	12.3747	16.0801
Standard deviation( $\sigma$ )	0.5029	1.8515	0.9072	1.0867	2.3154
Mean deadspace, $\mu$ (%)	2.0715	7.1286	7.6889	12.674	18.496
Average Time,t (s)	425.64	457.35	502.53	1719.07	3267.45
Number of Generations, Gen	100	100	100	130	150
Number of Populations, Pop	100	100	100	200	250

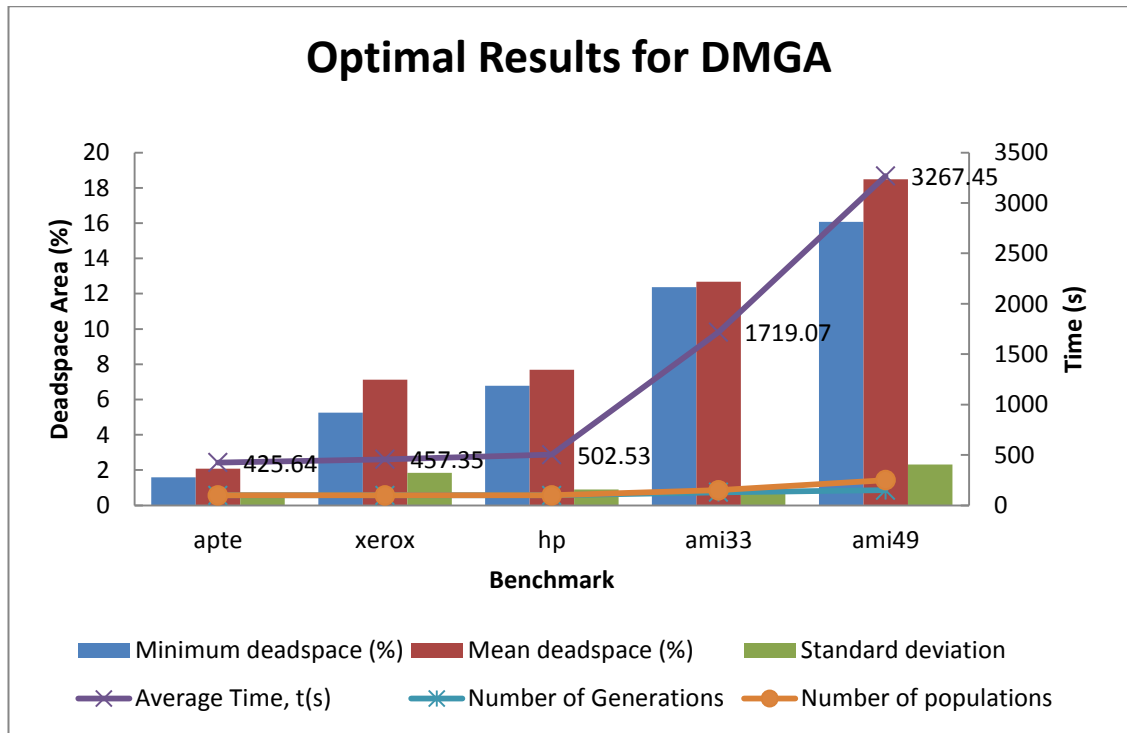


Figure 4.5 Graph showing the optimal results for DMGA

As the number of block increases, the deadspace percentage and time taken for optimization increases. This is because it is more cumbersome to achieve compact block placement based on the complexity of the sequence. Hence the probability to obtain the ideal answer reduces. Besides that, it is observed that the standard deviation increases as the number of blocks increases. As the number of blocks increases, wider range of optimization results will be obtained as there is a wider range of solution strings.

The population size and number of generation are selected based on the number of blocks. As the number of blocks increases, the generation and also population should be increased to produce more random generated strings so that higher chances to obtain better solution strings. Increase in number of blocks also will increase the time required to optimize the floorplan. From the table and graph, we can see that longer time is taken to optimize ami49 because more blocks are needed for placement and also the number of generation and population size for GA need to be increased. Below are the best results obtained for the benchmarks:

1. Apte – The minimum deadspace area obtained is 1.5887%. Figure 4.6 shows the most optimum placement for apte which has 9 blocks using GA as optimization with DM as representation.

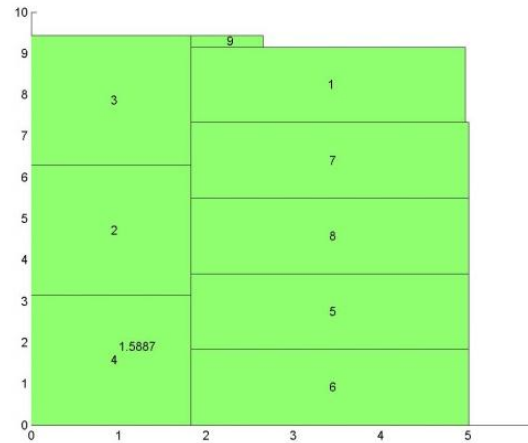


Figure 4.6 Placement for Apte

2. Xerox – The minimum deadspace area obtained is 5.2598%. Figure 4.7 shows the most optimum placement for xerox which has 10 blocks using GA as optimization with DM as representation.

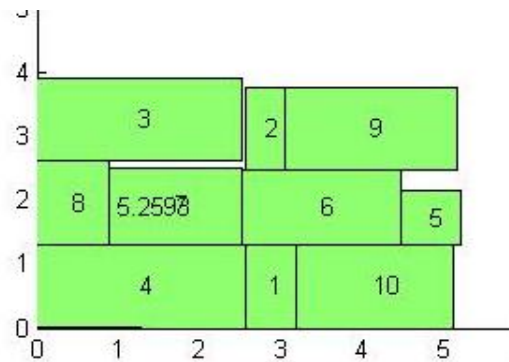


Figure 4.7 Placement for Xerox

3. Hp – The minimum deadspace area obtained is 7.9102%. Figure 4.8 shows the most optimum placement for hp which has 11 blocks using GA as optimization with DM as representation.

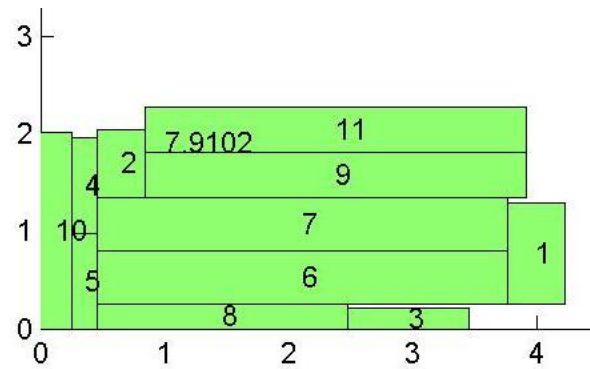


Figure 4.8 Placement for hp

4. Ami33 - The minimum deadspace area obtained is 12.3747%. Figure 4.9 shows the most optimum placement for ami33 which has 33 blocks using GA as optimization with DM as representation.

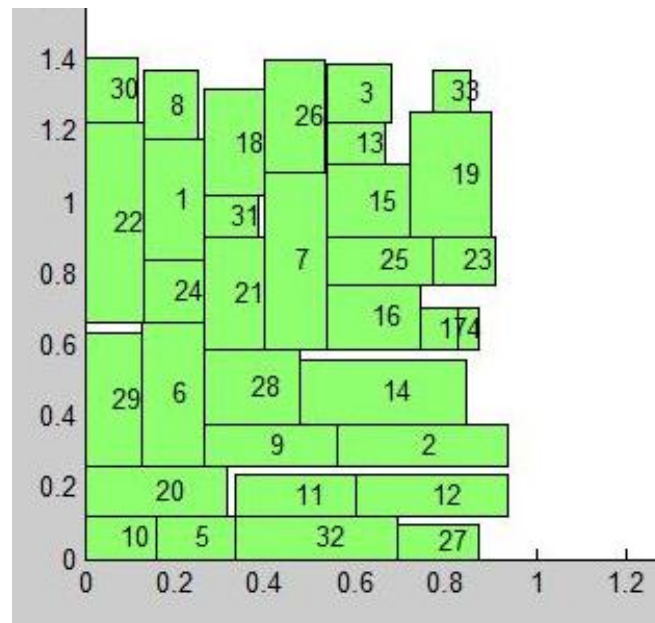


Figure 4.9 Placement for ami33

5. Ami49 - The minimum deadspace area obtained is 16.0801%. Figure 4.10 shows the most optimum placement for ami49 which has 49 blocks using GA as optimization with DM as representation.



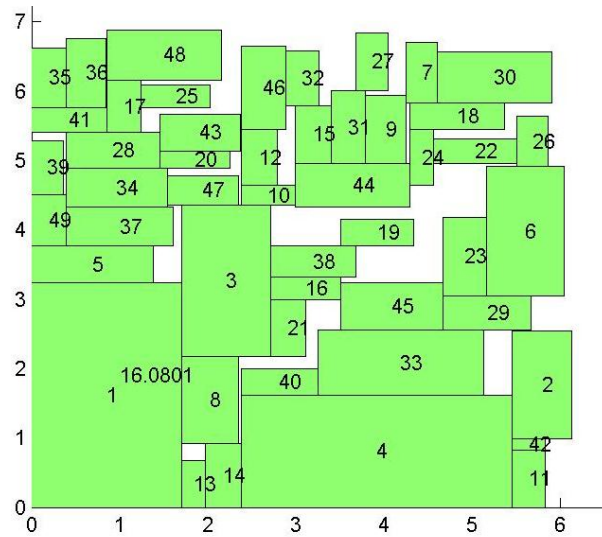


Figure 4.10 Placement for ami49

## 4.2 CBLL-GA

In the initial part of this section, discussion on the effects of mutation operators and crossover operators are discussed for binary representation and also ordered base sequence. This is to select the optimal frequency for mutation and crossover operators. The final section discuss on the optimal results for CBLL-GA.

### 4.2.1 Effects of mutation operators on binary sequence

In this section, the effects of mutation operators frequency for floating point representation is studied using population size of 100 for 100 generations. Benchmark xerox is used for this purpose. The floating point representation consists of 2 strings which are block position and the rotation of the block. The frequency of mutation is varied from low value (5) to high value (30) for each type of mutation operator. The mutation operators that were used are inversion mutation and binary mutation. Table 7 and Figure 4.11 show the optimization results for deadspace area and time when the frequency of mutation operators are varied for floating point representation.

Table 7: Study on the Frequency of Mutation Operators

Mutations Frequency		Deadspace(%)	Time (s)
Inversion	Binary		
5	5	2.93	4.2081
30	5	2.85	5.7212
5	30	2.95	6.4825
30	30	2.34	8.1587

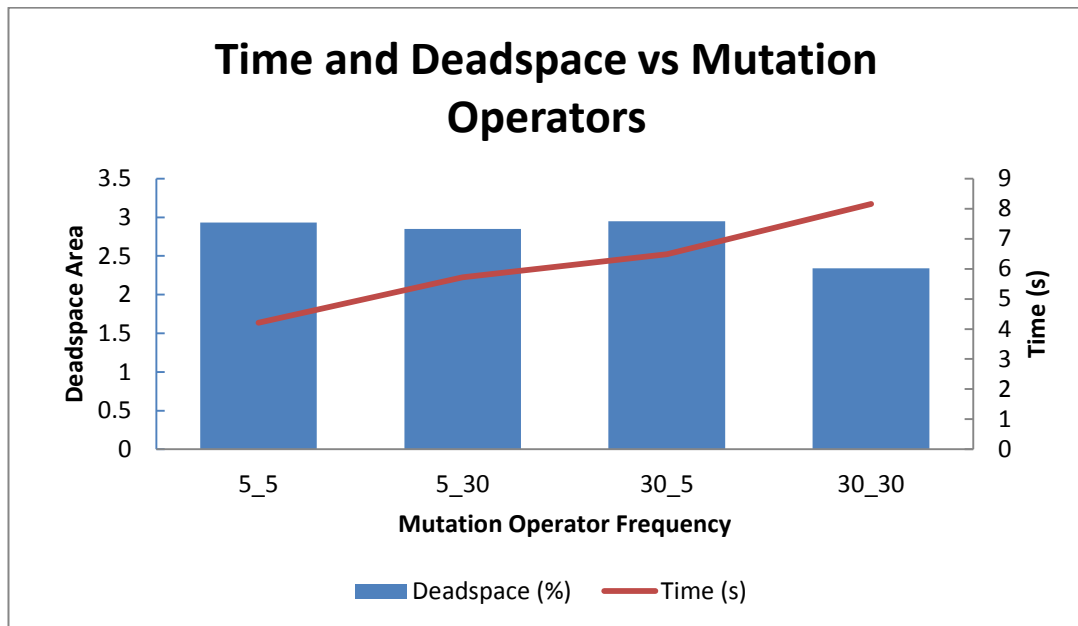


Figure 4.11 Study on Number of Mutation Operators for floating point representation

The result obtained is based on average results of 30 simulations for each case. From the results, we can observe that when all the mutation operations frequencies are high (30) for each mutation, the deadspace area obtained is the lowest. However, a longer time is needed to accommodate higher frequency of mutation operation. Nevertheless, the simulation takes less time as there are only two types of mutation involves in this optimization.

When the inversion mutation frequency is high (30), the deadspace area is slightly lower than when all the mutation frequency is low (5) and the time taken is

slightly longer in comparison to condition where all the mutation operations frequencies are low. When the binary mutation frequency is high (30), there is no significant difference in the deadspace area but the time taken is longer. However, when both mutation frequencies are high, the lowest deadspace area is obtained. From the observation, the frequency of the mutation operators that are selected is as follows:

- a) Inversion Mutation - 30
- b) Binary Mutation - 30

#### **4.2.2 Effects of crossover operator for binary representation**

In this section, the effect of crossover operator frequency for binary representation is studied with a population size of 100 for 100 generations. The benchmark used for this study is xerox. The binary representation consists of 2 different strings which are block position and rotation of the block. The frequencies of crossover operators are varied from low value (5) to high value (30). The crossover operators used is called simple crossover. Table 8 and Figure 4.12 show the optimization results for deadspace area and time when the frequency of crossover operators are varied for binary representation.

Table 8: Study on the Frequency of Crossover

Simple Crossover Frequency	Deadspace(%)	Time (s)
5	4.5189	4.2181
30	4.2181	4.2181

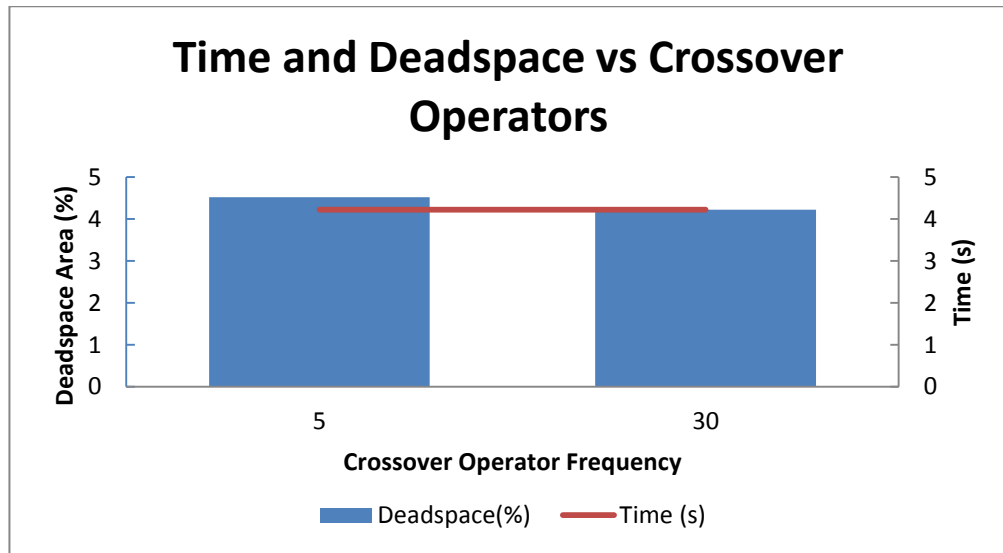


Figure 4.12 Study on the Number of Crossover for Floating Point Representation

The result obtained is based on mean results of 30 simulations for each case study. From the results, it can be observed that there is no significant difference in the deadspace area when the frequency for simple crossover is high (30) or low (5). Besides that, there is not much difference in the time taken to complete a simulation. It can be concluded that simple crossover frequency for binary representation does not have any significant effects on the deadspace area as well as time needed to complete a simulation.

#### 4.2.3 Effects of mutation operators for ordered based sequence

In this section, the effects of mutation operators for ordered based sequence is studied with a population size of 100 for 100 generations. The benchmark used for this study is xerox. The ordered based sequence used is for the current block placement. Hence this will affect the sequence where the blocks are chosen. The effects of the mutation operators which are studied are inversion mutation, adjacent swap mutation, shift mutation, swap mutation and threeswap mutation. The mutation operations frequencies are varied from low value (5) to high value (30). Table 10 and Figure 4.13

show the optimization results for deadspace area and time when the frequency of mutation operators are varied for ordered based sequence.

Table 9: Study on the Frequency of Mutation

Mutations Frequency					Deadspace (%)	Time (s)
Inversion	Adjacenet Swap	Shift	Swap	Threeswap		
5	5	5	5	5	2.93	4.0542
30	5	5	5	5	2.95	4.9214
5	30	5	5	5	3.04	5.0185
5	5	30	5	5	2.51	4.2401
5	5	5	30	5	2.97	5.2618
5	5	5	5	30	2.39	5.2751
30	30	30	30	30	2.89	8.5187

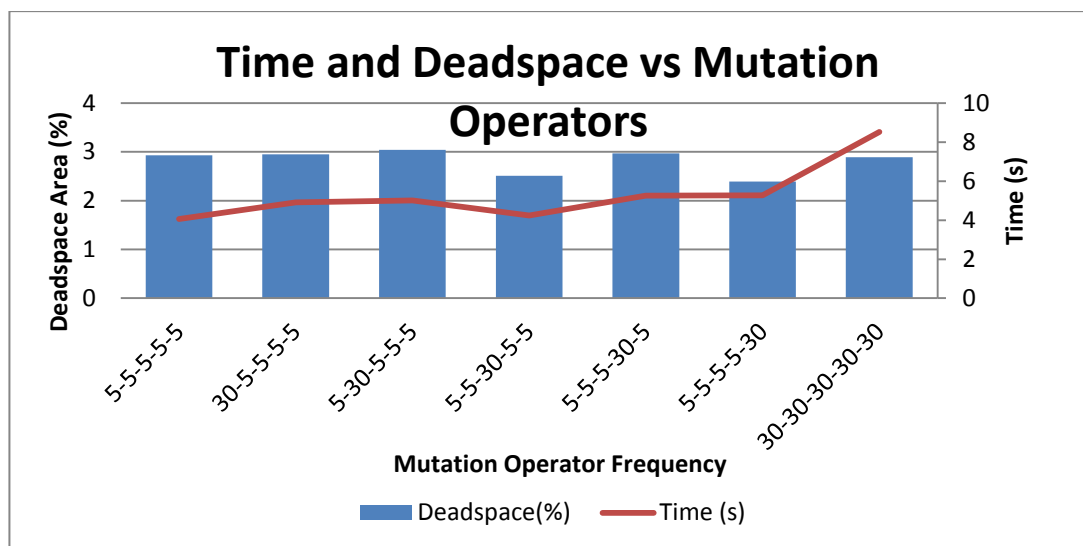


Figure 4.13 Study on the Frequency of Mutation for ordered based number

The result obtained is based on average results of 30 simulations for each case study. From the results, it can be observed that when the frequency for shift mutation or threeswap mutation is high (30), the deadspace area is the lowest. Time taken is only slightly longer than when all the frequencies of the mutation operations are low (5).

When the frequency of mutation operation for inversion mutation, adjacent swap mutation or swap mutation is high (30), there is not much change in the deadspace area

and only a slight increase in the time taken to complete one simulation. When all the frequencies of mutation operations are high, there is not much difference in the deadspace area and the time taken to complete a simulation is twice compared to the time taken when all the mutation operations frequencies are low. It can be concluded that only threeswap mutation should have higher frequency. Hence, the optimal frequency of crossover operator that is selected is as follow:

- a. Inversion Mutation – 5
- b. Adjacent Swap Mutation – 5
- c. Shift Mutation – 5
- d. Swap Mutation – 5
- e. Threeswap Mutation – 30

#### **4.2.4 Effects of crossover operators for ordered based number**

In this section, the effects of crossover operations for ordered based numbers is studied with population size of 100 for 100 generations. The benchmark used for this study is xerox. Ordered based sequences are used for the current block placement. Hence, this will affect the sequence where the blocks are chosen. The effects of the crossover operators that are studied are cyclic crossover, single point crossover, order-based crossover, uniform crossover and partial mapping crossover. The frequencies of crossover operations are varied from low value (5) to high value (30). Table 10 and Figure 4.14 show the effects in the optimization results for deadspace area and time when the frequency of crossover operators are varied for ordered based sequence.

Table 10: Study on the Frequency of Crossover

Crossover Frequency					Deadspace (%)	Time (s)
Cyclic	Single Point	Order-based	Uniform	Partial Mapping		
5	5	5	5	5	2.93	4.1054
30	5	5	5	5	2.67	4.0021
5	30	5	5	5	3.15	4.2105
5	5	30	5	5	2.88	4.5617
5	5	5	30	5	2.47	3.9897
5	5	5	5	30	3.14	4.0158
30	30	30	30	30	2.93	5.0464

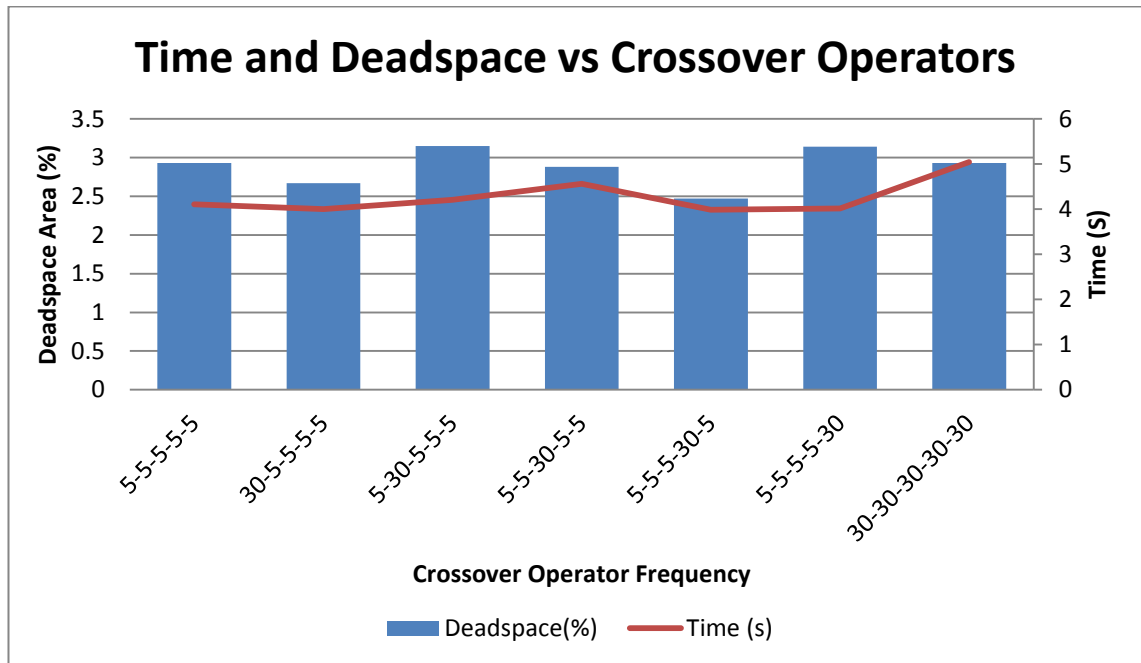


Figure 4.14 Study on the Frequency of Crossover for ordered based number

The result obtained is based on the average results of 30 simulations for each case. From the results, when only uniform crossover frequency is high (30) and the rest are retained low (5), the deadspace area obtained is the smallest. The time taken is also shorter compared to the time taken when all the crossover frequency are low.

When only cyclic crossover frequency is high (30), the deadspace area is second most minimum though the time taken is slightly lower than when only the uniform crossover frequency is high (30). The deadspace area and the time taken are almost the same when order based crossover and partial mapping crossover is high. However, when single point crossover frequency is set as high (30), the deadspace area increases although the time taken to complete the simulation is about the same as when all the crossover frequency is retained low. There is hardly any difference in the deadspace area when all the frequency of the crossover operations are retained high (30) or low (5) but the time taken to complete a simulation is the longest when all the crossover operators frequencies are high (30). It can be concluded that uniform crossover and cyclic crossover can have higher frequency in order to reduce the deadspace area without prolonging time taken to complete a simulation. Hence, the optimal frequency of crossover operator that is selected is as follow:

- a. Cyclic Crossover – 5
- b. Single Point Crossover – 5
- c. Order-based Crossover – 5
- d. Uniform Crossover – 30
- e. Partial Mapping Crossover - 5

#### **4.2.5 Optimal Results and Data Analysis**

For CBLL-GA, analyses had been carried out to test validity of the work. Table 11 and Figure 15 summarises the results of the benchmark for apte, hp, xerox, ami33 and ami49.



Table 11: Optimal Results for CBLL-GA

Benchmark	apte	xerox	hp	ami33	ami49
Number of blocks, n	9	10	11	33	49
Minimum deadspace, (%)	0.7697	2.479	1.318	2.201	2.690
Standard deviation( $\sigma$ )	0	0.3396	0	0.4114	0.5446
Mean deadspace, $\mu$ (%)	0.7697	2.479	1.318	2.448	3.555
Average Time, t (s)	1.0231	4.2451	5.1520	10.3215	18.4271
Number of Generations, Gen	100	100	100	130	150
Number of Populations, Pop	100	100	100	200	250

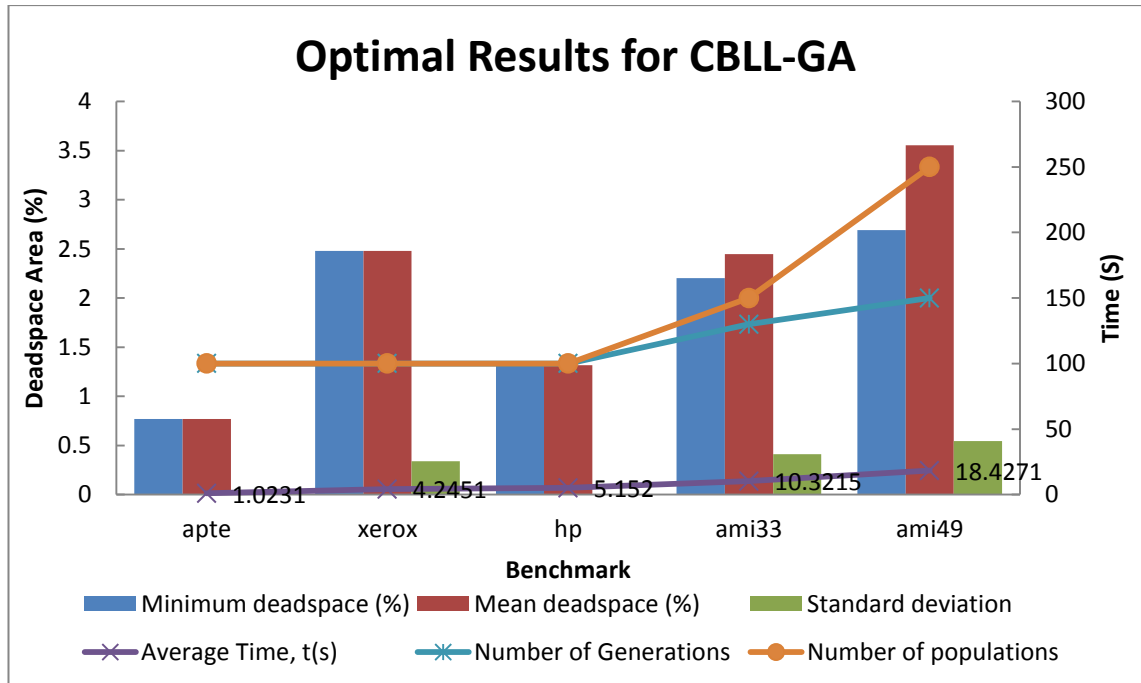


Figure 4.15 Graph showing the optimal results for CBLL-GA

As the number of block increases, the deadspace percentage and time taken for optimization increases. This is because it is more cumbersome to achieve compact block placement based on the complexity of the sequence and therefore increases the range of solution strings. Hence, the probability to obtain the optimum deadspace area reduces.

The population size and number of generation are selected based on the number of blocks. As the number of blocks increases, the generation and also population should

be increased to produce more random generated strings so that higher chances to obtain better solution strings. Increase in number of blocks also will increase the time required to optimize the floorplan. From the table and graph, the shortest time taken for a complete simulation is apte and the longest time taken to complete a simulation is ami49 because more blocks are needed for placement and also the number of generation and population size for GA needs to be increased. Following are the best results obtained for the benchmarks:

1. Apte – The minimum deadspace area obtained is 0.7697%. Figure 4.16 shows the most optimum placement for apte which has 9 blocks using GA as optimization with CBLL as representation.

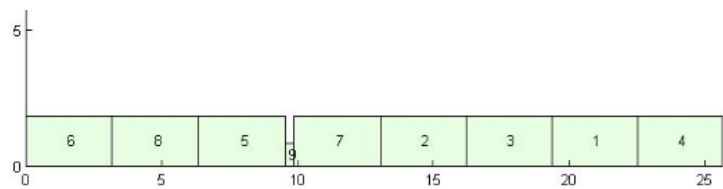


Figure 4.16 Placement for Apte

2. Xerox – The minimum deadspace area obtained is 2.479%. Figure 4.17 shows the most optimum placement for xerox which has 10 blocks using GA as optimization with CBLL as representation.

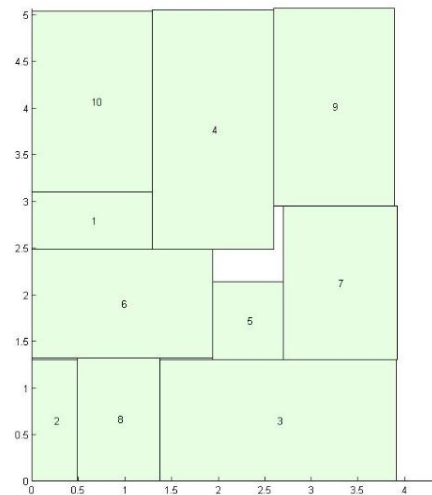


Figure 4.17 Placement for Xerox

3. Hp – The minimum deadspace area obtained is 1.318%. Figure 4.18 shows the most optimum placement for hp which has 11 blocks using GA as optimization with CBLL as representation.

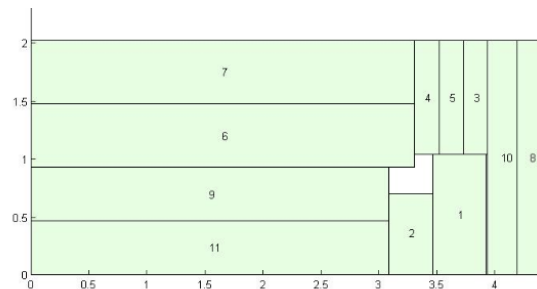


Figure 4.18 Placement for hp

4. Ami33 - The minimum deadspace area obtained is 2.201%. Figure 4.19 shows the most optimum placement for ami33 which has 33 blocks using GA as optimization with CBLL as representation.

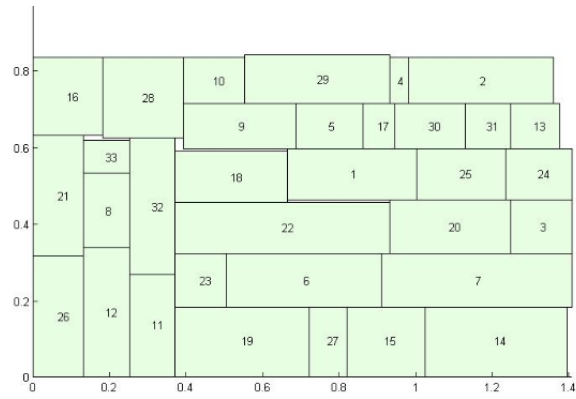


Figure 4.19 Placement for ami33

5. Ami49 - The minimum deadspace area obtained is 2.690%. Figure 4.20 shows the most optimum placement for ami49 which has 49 blocks using GA as optimization with CBLL as representation.

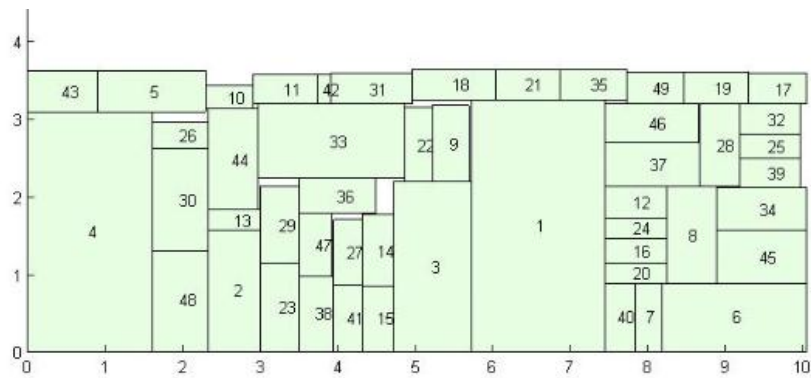


Figure 4.20 Placement for ami49

### 4.3 CBLL-CE

For CBLL and CE, a few analyses had been done in order to test validity of the work. Table 12 and Figure 4.21 summarises the results of the benchmark for apte, hp, xerox, ami33 and ami49.

Table 12: Optimal Results for CBLL and CE

Benchmark	apte	xerox	hp	ami33	ami49
Number of blocks, n	9	10	11	33	49
Minimum deadspace, (%)	0.7697	2.479	1.318	1.838	2.617
Standard deviation( $\sigma$ )	0	0.2159	0	0.2115	0.5422
Mean deadspace, $\mu$ (%)	0.7697	2.778	1.318	2.057	3.337
Average Time, t (s)	1.3057	10.247	24.957	204.521	375.275
Number of Sample, N	810	1000	1210	10890	24010

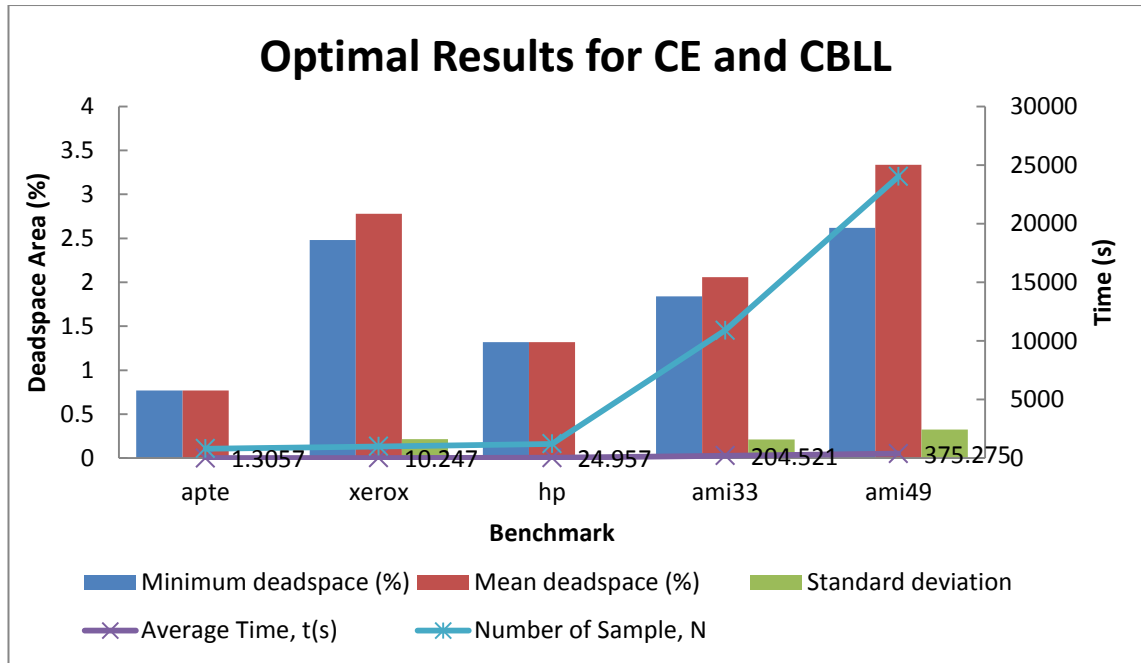


Figure 4.21 Graph showing the optimal results for CBLL and CE

The more number of block increases, time taken increases because more samples needed to be decoded in order to obtain the deadspace area of each sequence generated. Therefore, the time taken to complete an optimization increases exponentially when the number of block is increased by an exponential number. The reason to increase the representation string is so that more probability relationship can be generated in order to obtain the best deadspace between two blocks of the sample of the blocks to be optimized. This is because CE uses local deadspace probability in order to choose the placement of the blocks. Below are the best results obtained for the benchmarks:

1. Apte – The minimum deadspace area obtained is 0.7697%. Figure 4.22 shows the most optimum placement for apte which has 9 blocks using CE as optimization with CBLL as representation.

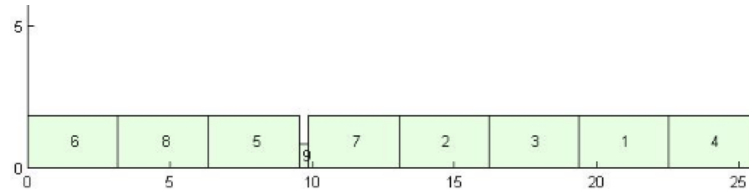


Figure 4.22 Placement for Apte

2. Xerox – The minimum deadspace area obtained is 2.479%. Figure 4.23 shows the most optimum placement for xerox which has 10 blocks using CE as optimization with CBLL as representation.

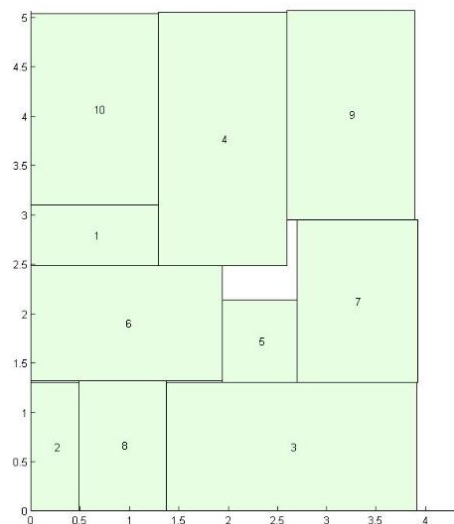


Figure 4.23 Placement for Xerox

3. Hp – The minimum deadspace area obtained is 1.318%. Figure 4.24 shows the most optimum placement for hp which has 11 blocks using CE as optimization with CBLL as representation.

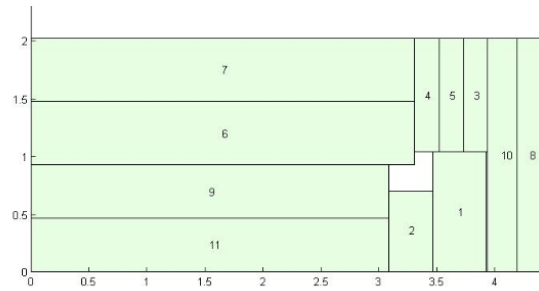


Figure 4.24 Placement for hp

4. Ami33 - The minimum deadspace area obtained is 1.838%. Figure 4.25 shows the most optimum placement for ami33 which has 33 blocks using CE as optimization with CBLL as representation.

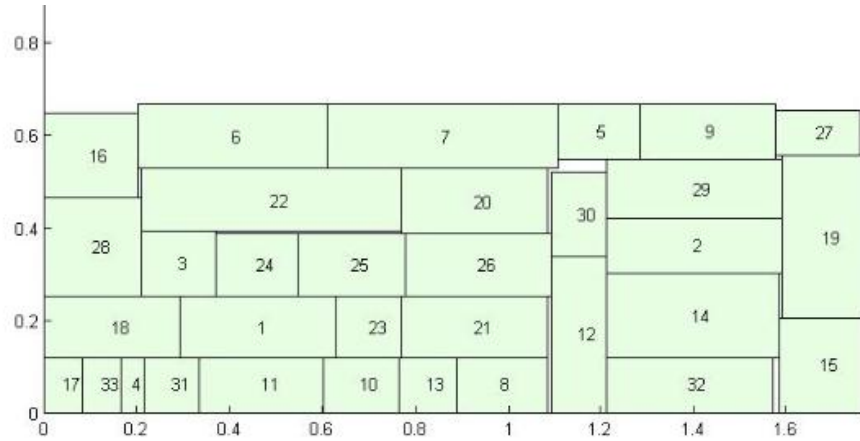


Figure 4.25 Placement for ami33

5. Ami49 - The minimum deadspace area obtained is 2.617%. Figure 4.26 shows the most optimum placement for ami49 which has 49 blocks using CE as optimization with CBLL as representation.

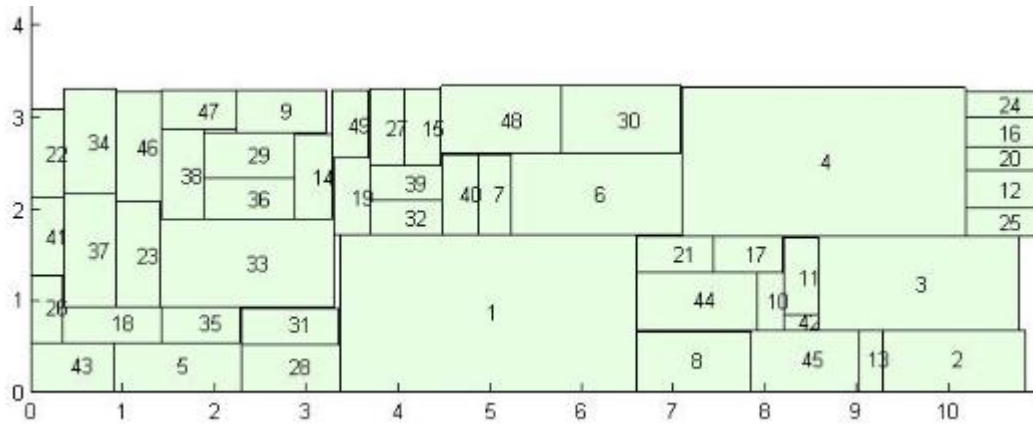


Figure 4.26 Placement for ami49

#### 4.4 Discussion

Comparing the three methods that were developed, it can be observed that DM takes a longer runtime to decode the representation into placement. DM is a more complicated representation and depends more on the optimization algorithm representation. In floorplanning optimization, we need to consider both time taken for decoding and also time taken to complete an optimization to obtain optimal results. Hence, CBLL is developed in order to reduce the time taken for decoding the representation.

Decoding time is important because we will need to decode the representation in order to obtain the deadspace area for a particular representation generated by the optimization algorithm. Hence, many deadspace area computation are needed from the floorplans representation strings in order to find the minimal deadspace area.

In CBLL representation, local deadspace area is determined according to the shapes determined by the contour which were chosen. The minimum space will be closed and void from other blocks to be placed in order to determine the shape of the contour. This is to minimize the space lost. Due to the limitations of the contour shape used in this present work, there is a limited placement combination that reduces search



space. This encourages very fast optimization but prevents the search for minimal deadspace area placement. The current work consists of 12 contour shapes and 4 points reference points are used for block placement.

The deterministic section of CBLL is where the least minimal space is being locked to determine the contour after placing the block so that the next block can be placed referring to the previous contour shape and points. This enables the reduction in local deadspace area before determining the deadspace area of the whole placement.

GA is a global search optimization technique. Hence, only the best solution strings are kept without considering the local deadspace area between two blocks. GA is a faster optimization technique compared to CE. However, the search for optimum point in GA is very random. This is because GA commences with a randomized population of placement. From this population, the best quantile will be selected to be bought to the next generation. Hence, the results of the optimization depend strongly on the first generation. This results in a very wide range of placements outcome for every optimization runs. This will cause higher standard deviation in the deadspace area results for GA compared to CE.

Mutation and crossover are randomly executed chromosomes to obtain fitter genes for the next generation. Mutation and crossover have very little impact on the local deadspace area unlike in CE. According to the GA, better parents will produce better child in next generation by undergoing crossover or mutation. This causes the optimization to rely on directed randomness.

GA is used to optimize the floorplanning representation of DM and CBLL because of its flexibility that can be easily modified. The genes in GA are easily modified so that it can represent for both DM and also CBLL. GA solution string, which is also the gene, can be represented either as order-based gene, floating point gene and

also binary gene. A gene can also be modified to have the combination of these different types of representation. For DM, the gene consists of a string of order-based representation and 2 strings of floating point representations. For CBLL, the gene consists of a string of order-based representation and 2 strings of binary representations.

CE is a local search optimization technique where it uses probability to obtain an optimal solution. After exhaustive search in literature, CE is nowhere found to be used in floorplanning optimization previously. This encourages the present study to attempt the capability of this new optimization algorithm on floorplanning. Similar to GA, CE is initialized with random generated variables. From these random generated variables, we are able to obtain the relationship between 2 different blocks during placement. This is because CE is able to measure the probability of adjacent block placement based on deadspace. The lower the deadspace area between the two blocks, the higher the probability of choosing this relationship between the two blocks. These probabilities are obtained during the updating of the transition matrix in the probability density function to determine the relationship between the blocks.

As CE strongly depends on the relationship between the blocks, the representation that can be used in CE is limited. Due to the complexity of DM representation, it will be complicated to modify the representation to suit CE and will take a longer time for optimization. Hence, only CBLL is used as a representation when using CE as optimization tool. This is because CBLL is a simpler representation and therefore it is easily modified to match CE algorithm. The CBLL is modified so that it can be placed in a three dimension matrix for CE transition matrix. The modification of the representation for CBLL to incorporate with CE is shown in chapter 3.3.1.2.

CE optimization algorithm gives better result compared to GA as it strongly depends on the relationship between two different blocks. Many representation strings

were initiated in algorithm. This is to increase the number of solution strings to obtain more accurate results. Hence, more accurate probability to obtain the best placement between two blocks is computed. Minimizing the local deadspace area will indirectly reduce the deadspace area of the total placement. Hence, CE gives better results for deadspace area. However, CE consumes long runtime to complete the simulation. This is because of the updating process of the transition matrix which consumed considerable amount of runtime.

In floorplan optimization, both runtime and deadspace area are considered when selecting the best optimization method. The target of floorplan optimization is to be able to obtain an optimal result in a short time. Hence, more work need to be done to improve the runtime of CE algorithm used for floorplanning optimization.s

Table 13 shows the MCNC Benchmarks which were used to compare and verify the results of DMGA, CBLL-GA and CBLL-CE and other methods which were developed previously.

Table 13: MCNC Benchmark Comparison

	apte		xerox		hp		ami33		ami49	
	Deadspace Area (%)	Time (s)	Deadspace Area (%)	Time (s)	Deadspace Area (%)	Time (s)	Deadspace Area (%)	Time (s)	Deadspace Area (%)	Time (s)
O-Tree	1.156	38	3.874	118	4.297	57	9.090	1430	6.079	7428
B*-Tree	0.7697	7	2.479	25	1.318	55	9.819	3417	3.822	4752
CS	0.7697	1	2.479	54	1.318	6	2.036	530	2.355	851
FAST-SP	0.7697	1	2.324	14	1.318	6	4.198	20	2.975	31
DMGA	1.589	425	5.259	457	6.778	503	12.375	1719	16.080	3267
CBLL-GA	0.7697	1	2.479	<b>4</b>	1.318	<b>5</b>	2.201	<b>10</b>	2.690	<b>18</b>
CBLL-CE	<b>0.7697</b>	1	<b>2.479</b>	10	<b>1.318</b>	25	<b>1.838</b>	204	2.617	375

Table 13 compares our results with other floorplan representations (O-Tree [1], B\*-Tree[2], CBL[3], CS[5] and also Fast-SP[4]) which were modelled by previous researchers using hard block MCNC benchmark circuits. The results show that DMGA does not give an optimal result compared to the other two methods which has been developed which are CBLL-GA and CBLL-CE. The results show that the deadspace

obtained from CBLL-GA is acceptable compared to existing algorithms. More significantly, CBLL-GA runtime is much shorter compared to others. The improved performance of the proposed CBLL representation is due to the introduction of multiple contour shapes for placement. CBLL-GA gives marginally larger deadspace area as compared to CS. CBLL has a limited number of reference points which reduces the solution space complexity. Thus CBLL-GA could generate compact floorplans in much shorter runtime.

From the results, the deadspace area obtained using CBLL-CE for *ami33* benchmark is lower compared to previously developed algorithms. The improved results are due to the fact that CE can be used to calculate the probabilities of the blocks relationships which enables the selection of the best pair of relationship between the blocks.

## CHAPTER 5. CONCLUSION AND FUTURE WORK

### 5.1 Conclusion

Floorplanning optimization consists of two sections namely the representation section and also the optimization algorithm section. Hence, the study was carried out on these two sections. The floorplan representation models developed are DM and CBLL. DM was combined with GA to execute floorplanning optimization whereas CBLL is embedded in both GA and CE to perform floorplanning optimization.

DM representation can represent many different shapes and therefore cover larger search space in comparison to CBLL. However, DM uses longer runtime to decode from representation strings to floorplan as compared to CBLL. CBLL is a representation where it has a limited number of shapes of contour which will reduce the search space compared to DM. However, CBLL has an embedded deterministic algorithm which will calculate and choose the smallest local deadspace area during placement compared to DM which solely depends on optimization tool as in this case, GA.

GA was used in many previous floorplanning studies. GA is much more flexible compared to CE. This is because the chromosomes in GA can be modified easily to suit the representation used. GA takes a shorter runtime for optimization as it uses mutation and crossover operations on some of the best quantile population to be brought to the next generation. Not all the chromosomes are involved in the mutation and crossover operations. CE takes a much longer time to complete a simulation as it considers all the solution strings which were generated and is used for the transition matrix in order to obtain better results in the subsequent iteration.

CE optimization algorithm gives better results comparatively to GA. CE advantage relies on detailed relationship between two adjacent blocks. Hence, CE gives

smaller deadspace area. However, CE requires a longer runtime to complete the simulation as the updating process of the transition matrix consumes considerable time.

In floorplan optimization, both time and deadspace area need to be considered. The target of floorplan optimization is to be able to obtain an optimal result in a short time. Hence, more work need to be done in order to improve the CE algorithm so that to complete a simulation completes in a shorter time.

CBLL reduces the time taken for decoding from representation to floorplan. This greatly simplifies and reduces the runtime of the optimization. This is because CBLL reduces the complexity space of floorplan placement due to the limited number of contour shapes in the CBLL representation. In the present work, the time taken for optimization is reduced and also at the same time sustaining the deadspace area obtained. Substantial improvement in the runtime ratios compared to other models is observed as the number of blocks increases.

CBLL-CE, a new algorithm, is proposed in this research to incorporate CE to optimize floorplanning problems. The CE method is developed using cross entropy distance which is known as the Kullback-Leibler distance. This method is motivated by an adaptive algorithm to estimate probabilities in rare events involving minimization. CBLL-CE gives a better result in terms of deadspace area for floorplan optimization as compared to the method we have used previously which is CBLL representation involving GA. Also CBLL-CE gives improved performance for previously developed floorplan representations using both EA and SA. The improved performance of using CE is based on the detailed relationship between two adjacent blocks and this will give a better deadspace area when optimization is performed. It is concluded that the proposed CE method can be used to implement VLSI floorplan optimization with reduction in the deadspace area of the floorplan when the number of modules increases.

In conclusion, two floorplanning representation models and two optimization algorithm were reviewed and were compared with the other methods which were used previously. We can conclude that CE-CBLL gives minimum deadspace area for floorplanning optimization compared to previously developed method and GA-CBLL gives a substantial improvement in the optimization runtime without affecting the deadspace area.

## **5.2 Future Work**

To improve the floorplan optimization, more detailed work needs to be done. In order to improve the results for DM and CBLL, GA algorithm needs to be modified so that the mutation and crossover operations can be used effectively to execute the placement of the blocks instead of entirely random operation. This is to enable GA to optimize according to DM and CBLL representations. Since CBLL is faster compared to DM, more improvements should be worked on to improve the results for CBLL-GA.

In order to improve the results for CBLL-CE, we need to improve the CBLL representation so that more contour shapes will be taken into consideration to reduce the amount of local space during packing according to the algorithm. This is to reduce the final deadspace area of the placement. A better programming needs to be implemented to improve the time taken for a complete simulation using CE method. This is because CE uses a long runtime especially during updating of the transition matrix. Perhaps parallel computation programming needs to be implemented in order to speed up the simulation time for CBLL-CE.

In conclusion, the present study lays a foundation for new floorplanning representation such as DM and CBLL. Besides that, the present work explores the possibility of CE as optimization tool in floorplanning which had not been implemented in floorplanning optimization previously.

## REFERENCES

- Alpert, C. J., Mehta, D. P., & Sapatnekar, S. S. (2009). *Handbook of Algorithms for Physical Design Automation*. Boca Raton: Taylor & Francis Group.
- Chang, Y.-C., Chang, Y.-W., Wu, G.-M., & Wu, S.-W. (2000). B\*-trees: A New Representation for Non-slicing Floorplans. *Annual ACM IEEE Design Automation Conference* (pp. 458-463). Los Angeles: ACM.
- Chen, D.-S., Lin, C.-T., & Wang, T.-W. (2003). Non-slicing floorplans with boundary constraints using generalized Polish expression. *Design Automation Conference 2003. Proceedings of the ASP-DAC 2003*, (pp. 342-345).
- Chen, J., & Zhu, W. (2010). A hybrid Genetic Algorithm for VLSI Floorplanning. *Intelligent Computing and Intelligent Systems (ICIS), 2010 IEEE International Conference on*, (pp. 128-132).
- Chen, J., Zhu, W., & Ali, M. (2010). A Hybrid Simulated Annealing Algorithm for Nonslicing VLSI Floorplanning. *Systems, Man and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on*.
- Chen, S., Dong, S., Hong, X., Ma, Y., & Cheng, C. (2006). VLSI Block Placement with Allignment Constraints. *Circuits and System-III Express Briefs, IEEE Transaction on*, 622-626.
- Chen, T.-C., & Chang, Y.-W. (2006). Modern Floorplanning Based on B\*-Tree and Fast Simulated Annealing. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*.
- Debarshi, C., & Manikas, T. W. (2007). A Genetic Algorithm for Non-Slicing Floorplan Representation. *National Conference on Intelligent Systems*. Hyderabad.



- Dhamdhere, S., Zhou, N., & Wang, T.-C. (2002). Modules Placement with Pre-Placed Modules using the Corner Block List Representation. *Circuit and Systems, 2002. ISCAS 2002. IEEE International Symposium on*, (pp. I-349 - I-352).
- Drakidis, A., Mack, R. J., & Massara, R. (2006). Packing-based VLSI module placement using genetic algorithm with sequence-pair representation. *Circuits, Devices and Systems, IEEE Proceedings*, 545-551.
- Guo, P.-N., Cheng, C.-K., & Yoshimura, T. (1999). An O-tree Representation of Non-slicing Floorplan and its Applications. *Annual ACM IEEE Design Automation Conference* (pp. 268-273). New Orleans: ACM.
- Holland, J. H. (1992). *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence*. Cambridge: MIT Press.
- Hong, X., Dong, S., Huang, G., Cai, Y., Cheng, C.-K., & Gu, J. (2004). Corner Block List Representation and Its Application to Floorplan Optimization. *Circuits and Systems II: Express Briefs, IEEE Transactions on*, 228-233.
- Hong, X., Huang, G., Cai, Y., Gu, J., Dong, S., Cheng, C.-K., et al. (2000). Corner block list: and effective and efficient topological representation of non-slicing floorplan. *Computer Aided Design, 2000. ICCAD-2000. IEEE/ACM International Conference on*, (pp. 8-12).
- Houck, C. R., Joines, J., & Kay, M. (n.d.). *GA source code*. Retrieved January 15, 2010, from GNU General Public License: <http://www.ise.ncsu.edu/mirage/GAToolBos/gaot/gaotindex.html>
- Jiang, Y.-H., Lai, J., & Wang, T.-C. (2001). Module Placement with Pre-placed Modules using the B\*-Tree Representation. *Circuits and Systems, 2001. ISCAS 2001. The 2001 IEEE International Symposium on*, vol 5, (pp. 347-350).

- Kodama, C., & Fujiyoshi, K. (2002). An Efficient Decoding Method of Sequence Pair. *Circuits and Systems, 2002. APCCAS '02. 2002 Asia Pacific Conference on*, (pp. 131-136).
- Kodama, C., & Fujiyoshi, K. (2003). Selected Sequence-Pair: An Efficient Decodable Packing Representation in Linear Time using Sequence Pair. *Design Automation Conference, 2003. Proceedings of the ASP-SAC 2003. Asia and South Pacific*, (pp. 331-337).
- Lai, M., & Wong, D. (2001). Slicing Tree is a Complete Floorplan Representation. *Design, Automation and Test in Europe 2001. Proceedings*, (pp. 228-232).
- Lin, C.-T., Chen, D.-S., & Wang, Y.-W. (2002). GPE: A New Representation for VLSI Floorplan Problem. *Computer Design: VLSI in Computers and Processors, 2002. Proceedings. 2002 IEEE International Congerence on*, (pp. 42-44).
- Lin, J.-M., & Chang, Y.-W. (2001). TCG: A Transitive Closure Graph-Based Representation for Non-slicing Floorplans. *Design Automation Conference, 2001, Proceedings*, (pp. 764 - 769).
- Lin, J.-M., & Chang, Y.-W. (2002). TCG-S: orthogonal coupling of P\*-admissible representations for general floorplans. *Design Automation Conference, 2002. Proceedings. 39th*, (pp. 842-847).
- Lin, J.-M., & Chang, Y.-W. (2004). TCG-S: orthogonal coupling of P\*-admissible representations for general floorplans. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, (pp. 968-980).
- Lin, J.-M., & Chang, Y.-W. (2005). TCG: A Transitive Closure Graph-Based Representation for General Floorplans. *IEEE Transaction on Very Large Scale Integration (VLSI) system, Vol 13, No. 2*, (pp. 288-292).

- Lin, J.-M., Chang, Y.-W., & Lin, S.-P. (2003). Corner sequence - a P-admissible floorplan representation with a worst case linear-time packing scheme. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, (pp. 679-686).
- Mao, F., Xu, N., & Ma, Y. (2009). Hybrid Algorithm for Floorplanning Using B\*-tree Representation. *Interlligent Information Technology Application, 2009. IITA 2009. Third International Symposium on*, (pp. 228-231).
- Mitchell, M. (1999). *An Introduction to Genetic Algorithms*. The MIT Press.
- Murata, H., Fujiyoshi, K., Nakatake, S., & Kajitani, Y. (1995). Rectangle-Packing-Based Module Placement. *Computer-Aided Design, 1995. ICCAD-95. Digest of Technical Papers., 1995 IEEE/ACM International Conference on*, (pp. 472-479).
- Nakatake, S., Fujiyoshi, K., Murata, H., & Kajiya, Y. (1996). Module Placement on BSG-Structure and IC Layout Applications. *Computer-Aided Design, 1996. ICCAD-96. Digest of Technical Papers., 1996 IEEE/ACM International Conference*, (pp. 484-491).
- Nakaya, S., Koide, T., & Wakabayashi, S. (2000). An Adaptive Genetic Algorithm For VLSI Floorplanning Based on Sequence-Pair. *Circuits and Systems, 2000. ISCAS 2000 Geneva. The 2000 IEEE International Symposium on*, (pp. 65-68).
- Ninomiya, H., Numayama, K., & Asai, H. (2006). Two-staged Tabu Search for Floorplan Problem Using O-Tree Representation. *Evolutionary Computation 2006. CEC 2006. IEEE Congress on*, (pp. 718-724).
- Rubinstein, R. Y., & Kroese, D. P. (2004). *The Cross-Entropy Method: A unified approach to Combinatorial Optimization, Monte-Carlo Simulation and Machine Learning*. New York: Springer Science + Business Media, Inc.
- Rubinstein, R. Y., & Kroese, D. P. (2008). *Simulation and the monte carlo method*. New Jersey: John Wiley & Sons, inc.

- Sait, S. M., & Youssef, H. (1999). *VLSI Physical Design Automation: Theory and Practice*. Singapore: World Scientific Publishing Co. Pte. Ltd.
- Sherwani, Naveed A. (2002). *Algorithms for VLSI Physical Design Automation*. Dordrecht: Kluwer Academic Publishers.
- Sitzmann, I., & Stuckey, P. (2000). O-Trees: a Constraint-based Index Structure. *Database Conference, 2000. ADC 2000 Proceedings, 11th Australasian*, (pp. 127-134).
- Sun, T.-Y., Hsieh, S.-T., Wang, H.-M., & Lin, C.-W. (2006). Floorplanning based on particle swarm optimization. *Emerging VLSI Technologies and Architectures, 2006. IEEE Computer Society Annual Symposium on*.
- Takahashi, T., Guo, P., Cheng, C., & Yoshimura, T. (2003). Floorplanning Using a Tree Representation: A Summary. *Circuit and Systems Magazine, IEEE*, 26-29.
- Tang, X., Tian, T., & Wong, D. (2001). Fast evaluation of sequence pair in block placement by longest common subsequence computation. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 1406-1413.
- Wong, D. F., & Liu, C. L. (1986). A New Algorithm For Floorplan Design. *In Proceeding ACM Design Automation Conference*, (pp. 101-107).
- Wu, M.-C., & Chang, Y.-W. (2004). Placement with Alignment and Performance Constraints Using the B\*-tree Representation. *Computer Design: VLSI in Computers and Processors, 2004. Proceeding. IEEE International Conference on*, (pp. 568-571).
- Xu, N., & Li, L. (2008). Hybrid Algorithm for Non-slicing Floorplans Optimization. *IEEE*.
- Yan, T., Li, J., Yang, B., & Yu, J. (2004). A Modified O-Tree Based Packing Algorithm and its Applications. *Communications, Circuits and Systems, 2004. ICCCAS 2004. 2004 International Conference on*, (pp. 1266-1270).

- Young, F., Wong, D., & Yang, H. (1999). Slicing Floorplan with Boundary Constraints. *IEEE Transaction on Computer-Aided Design of Integrated Circuits and Systems, Vol 18, No 9*, 1385-1389.
- Zhou, H., & Wang, J. (2004). ACG-adjacent constraint graph for general floorplans. *Computer Design: VLSI in Computers and Processors, 2004. ICCD 2004. Proceedings. IEEE Conference On*, (pp. 572-575).

## APPENDIX A

Matlab code for Genetic Algorithm function:

```
function [x,endPop,bPop,traceInfo] = ga(bounds,evalFN,termOps,termFN,mutFNs,mutOps,xOverFNs,xOverOps,...
    xOverFNsO,xOverOpsO,mutFNsO,mutOpsO,selectFN,selectOps,...
    evalOps)

n=nargin;
if n<2
    disp('Insufficient arguments')
end

epsilon = 1e-6; %Threshold for two fitness to differ
e1str=['c1(1:var) c1(xZomeLength)']=evalFN '(c1(1:var),numVars,[gen evalOps]);';
e2str=['c2(1:var) c2(xZomeLength)']=evalFN '(c2(1:var),numVars,[gen evalOps]);';
% termOps=[100];
% termFN=['maxGenTerm'];
% mutFNs=['boundaryMutation multiNonUnifMutation nonUnifMutation unifMutation'];
% mutOps=[4 0 0;4 termOps(1) 3;4 termOps(1) 3;4 0 0];
% xOverFNs=['arithXover heuristicXover simpleXover'];
% xOverOps=[2 0;2 3;2 0];
% xOverFNsO = ['cyclicXover linerorderXover singleptXover '...
%             'orderbasedXover partmapXover uniformXover'];
% xOverOpsO = [2;2;2;2;2;2];
% mutFNsO = ['inversionMutation adjswapMutation shiftMutation swapMutation threeswapMutation'];
% mutOpsO = [2;2;2;2;2];
% selectFN=['normGeomSelect'];
% selectOps=[0.08];
% termOps=[100];
% termFN='maxGenTerm';
% evalOps = [0 150];

% if isempty(startPop) %Generate a population at random
% startPop=zeros(80,size(bounds,1)+1);
% startPop=initializegenone(evalOps,bounds,evalFN);
% end

xOverFNs=parse(xOverFNs);
mutFNs=parse(mutFNs);
xOverFNsO=parse(xOverFNsO);
mutFNsO=parse(mutFNsO);

xZomeLength = size(startPop,2); %Length of the xzome=numVars+fitness
var = xZomeLength-1; %Number of variables\
numVars = bounds;
variable = var/3;
var1 = variable; %Block Number
var2 = 2*variable; %relative block number
var3 = 3*variable; %position of block
popSize = size(startPop,1); %Number of individuals in the pop
endPop = zeros(popSize,xZomeLength); %A secondary population matrix
c1 = zeros(1,xZomeLength); %An individual
c2 = zeros(1,xZomeLength); %An individual
numXOvers = size(xOverFNs,1); %Number of Crossover operators
numMuts = size(mutFNs,1); %Number of Mutation operators
numXOversO = size(xOverFNsO,1); %Number of Crossover operators order based
numMutsO = size(mutFNsO,1); %Number of Mutation operators order based
oval = max(startPop(:,xZomeLength)); %Best value in start pop
bFoundIn = 1; %Number of times best has changed
done = 0; %Done with simulated evolution
gen = 1; %Current Generation Number
```

```

collectTrace = (nargout>3);           %Should we collect info every gen
% floatGA    = opts(2)==1;           %Probabilistic application of ops
% display    = opts(3);             %Display progress

while(~done)
    %Elitist Model
    [bval,bindx] = max(startPop(:,xZomeLength)); %Best of current pop
    best = startPop(bindx,:);

    if collectTrace
        traceInfo(gen,1)=gen;           %current generation
        traceInfo(gen,2)=startPop(bindx,xZomeLength); %Best fitness
        traceInfo(gen,3)=mean(startPop(:,xZomeLength)); %Avg fitness
        traceInfo(gen,4)=std(startPop(:,xZomeLength));
    end

    if ( (abs(bval - oval)>epsilon) | (gen==1)) %If we have a new best sol
        fprintf(1,'\n%d %f\n',gen,bval) %Update the display
        bPop(bFoundIn,:)= [gen startPop(bindx,:)]; %Update bPop Matrix
        bFoundIn=bFoundIn+1; %Update number of changes
        oval=bval; %Update the best val
    else
        fprintf(1,'%d ',gen) %Otherwise just update num gen
    end

    endPop = feval(selectFN,startPop,[gen selectOps]); %Select

%for block number crossover
for i = 1:numXOversO,
    for j = 1:xOverOpsO(i,1),
        a = round(rand*(popSize-1)+1); %Pick a parent
        b = round(rand*(popSize-1)+1); %Pick another parent
        xN=deblank(xOverFNsO(i,:)); %Get the name of crossover function
        p1 = [endPop(a,1:var1),endPop(a,xZomeLength)];
        p2 = [endPop(b,1:var1),endPop(b,xZomeLength)];
        [C1 C2] = feval(xN,p1,p2,bounds,[gen xOverOpsO(i,:)]);
        c1(1:var1) = C1(1:variable);
        c1(var1+1:var3) = endPop(a,var1+1:var3);
        c2(1:var1) = C2(1:variable);
        c2(var1+1:var3) = endPop(b,var1+1:var3);

        if c1(1:var)==endPop(a,(1:var))
            c1(xZomeLength)=endPop(a,xZomeLength);
        elseif c1(1:var)==endPop(b,(1:var))
            c1(xZomeLength)=endPop(b,xZomeLength);
        else
            eval(e1str);
        end
        if c2(1:var)==endPop(a,(1:var))
            c2(xZomeLength)=endPop(a,xZomeLength);
        elseif c2(1:var)==endPop(b,(1:var))
            c2(xZomeLength)=endPop(b,xZomeLength);
        else
            eval(e2str);
        end
    end
    endPop(a,:)=c1;
    endPop(b,:)=c2;
end

% for relative block crossover
for i=1:numXOvers,
    for j=1:xOverOps(i,1),
        a = round(rand*(popSize-1)+1); %Pick a parent

```

```

        b = round(rand*(popSize-1)+1);          %Pick another parent
        xN=deblank(xOverFNs(i,:)); %Get the name of crossover function
    p1 = [endPop(a,var1+1:var2),endPop(a,xZomeLength)];
    p2 = [endPop(b,var1+1:var2),endPop(b,xZomeLength)];
        [C1 C2] = feval(xN,p1,p2,bounds,[gen xOverOps(i,:)]);
    c1(var1+1:var2) = round(C1(1:variable));
    c1(1:var1) = endPop(a,1:var1);
    c1(var2+1:var3) = endPop(a,var2+1:var3);
    c2(var1+1:var2) = round(C2(1:variable));
    c2(1:var1) = endPop(a,1:var1);
    c2(var2+1:var3) = endPop(a,var2+1:var3);

    if c1(1:var)==endPop(a,(1:var)) %Make sure we created a new
        c1(xZomeLength)=endPop(a,xZomeLength); %solution before evaluating
    elseif c1(1:var)==endPop(b,(1:var))
        c1(xZomeLength)=endPop(b,xZomeLength);
    else
        %[c1(xZomeLength) c1] = feval(evalFN,c1,[gen evalOps]);
        eval(e1str);
    end
    if c2(1:var)==endPop(a,(1:var))
        c2(xZomeLength)=endPop(a,xZomeLength);
    elseif c2(1:var)==endPop(b,(1:var))
        c2(xZomeLength)=endPop(b,xZomeLength);
    else
        %[c2(xZomeLength) c2] = feval(evalFN,c2,[gen evalOps]);
        eval(e2str);
    end

    endPop(a,:)=c1;
    endPop(b,:)=c2;

end
end

% for position crossover
for i=1:numXOvers,
    for j=1:xOverOps(i,1),
        a = round(rand*(popSize-1)+1);          %Pick a parent
        b = round(rand*(popSize-1)+1);          %Pick another parent
        xN=deblank(xOverFNs(i,:)); %Get the name of crossover function
    p1 = [endPop(a,var2+1:var3),endPop(a,xZomeLength)];
    p2 = [endPop(b,var2+1:var3),endPop(b,xZomeLength)];
        [C1 C2] = feval(xN,p1,p2,4,[gen xOverOps(i,:)]);
    c1(var2+1:var3) = round(C1(1:variable));
    c1(1:var1) = endPop(a,1:var1);
    c1(var1+1:var2) = endPop(a,var1+1:var2);
    c2(var2+1:var3) = round(C2(1:variable));
    c2(1:var1) = endPop(a,1:var1);
    c2(var1+1:var2) = endPop(a,var1+1:var2);

    if c1(1:var)==endPop(a,(1:var)) %Make sure we created a new
        c1(xZomeLength)=endPop(a,xZomeLength); %solution before evaluating
    elseif c1(1:var)==endPop(b,(1:var))
        c1(xZomeLength)=endPop(b,xZomeLength);
    else
        %[c1(xZomeLength) c1] = feval(evalFN,c1,[gen evalOps]);
        eval(e1str);
    end
    if c2(1:var)==endPop(a,(1:var))
        c2(xZomeLength)=endPop(a,xZomeLength);
    elseif c2(1:var)==endPop(b,(1:var))
        c2(xZomeLength)=endPop(b,xZomeLength);
    else
        %[c2(xZomeLength) c2] = feval(evalFN,c2,[gen evalOps]);
        eval(e2str);
    end
end
end

```



```

        endPop(a,:)=c1;
        endPop(b,:)=c2;
    end
end

%for block number mutation
for i = 1:numMutsO,
    for j = 1:mutOpsO(i,1),
        a = round(rand*(popSize-1)+1); %Pick a parent
        xN=deblank(mutFNsO(i,:)); %Get the name of crossover function
        p1 = [endPop(a,1:var1),endPop(a,xZomeLength)];
        C1 = feval(xN,p1,bounds,[gen mutOpsO(i,:)]);
        c1(1:var1) = C1(1:variable);
        c1(var1+1:var3) = endPop(a,var1+1:var3);
        if c1(1:var)==endPop(a,(1:var))
            c1(xZomeLength)=endPop(a,xZomeLength);
        else
            eval(e1str);
        end
    end
    endPop(a,:)=c1;
end

%for relative block mutation
for i=1:numMuts,
    for j=1:mutOps(i,1),
        a = round(rand*(popSize-1)+1);
        p1 = [endPop(a,var1+1:var2),endPop(a,xZomeLength)];
        c1 = feval(deblank(mutFNs(i,:)),p1,bounds,[gen mutOps(i,:)]);
        c1 = round(c1(1:variable));
        c1 = [endPop(a,(1:var1)), c1,endPop(a,(var2+1:var3))];
        if c1(1:var)==endPop(a,(1:var));
            c1(xZomeLength)=endPop(a,xZomeLength);
        else
            %[c1(xZomeLength) c1] = feval(evalFN,c1,[gen evalOps]);
            eval(e1str);
        end
        endPop(a,:)=c1;
    end
end

% end

%for blocks position
for i=1:numMuts,
    for j=1:mutOps(i,1),
        a = round(rand*(popSize-1)+1);
        p1 = [endPop(a,var2+1:var3),endPop(a,xZomeLength)];
        c1 = feval(deblank(mutFNs(i,:)),p1,4,[gen mutOps(i,:)]);
        c1 = round(c1(1:variable));
        c1 = [endPop(a,(1:var2)), c1];
        if c1(1:var)==endPop(a,(1:var))
            c1(xZomeLength)=endPop(a,xZomeLength);
        else
            %[c1(xZomeLength) c1] = feval(evalFN,c1,[gen evalOps]);
            eval(e1str);
        end
        endPop(a,:)=c1;
    end
end

gen=gen+1

```

```

done=feval(termFN,[gen termOps],bPop,endPop); %See if the ga is done
startPop=endPop; %Swap the populations

[~,bindx] = min(startPop(:,xZomeLength)); %Keep the best solution
startPop(bindx,:) = best; %replace it with the worst
end

[bval,bindx] = max(startPop(:,xZomeLength));
fprintf(1,'\n%d %f\n',gen,bval);

x=startPop(bindx,:);
x(1:var)
[~,~,xframe,yframe]=fitness(x(1:var),variable)
plotgraph(xframe,yframe,bounds)
bPop(bFoundIn,:)=startPop(bindx,:);
traceInfo(gen,1)=gen; %current generation
traceInfo(gen,2)=startPop(bindx,xZomeLength); %Best fitness
traceInfo(gen,3)=mean(startPop(:,xZomeLength)); %Avg fitness

%% -----Initialize first generation-----%%
function pop = initializegenone(option,bounds,evalFN)

% option = [ gen num_pop num_blocks]
%evalFN is fitness for this project
num = option(2);
numVars = bounds;
xZomeLength = 3*numVars+1; %Length of string is numVar + fit
var = xZomeLength - 1;
estr=['[pop(i,1:var) pop(i,xZomeLength) xframe yframe]= ' evalFN '(pop(i,1:var),numVars)'];
pop= zeros(num,xZomeLength);

for i = 1:num
    pop(i,1:3*numVars)=[randperm(numVars) (round(rand(1,numVars)*(numVars-1))+1)
    (round(rand(1,numVars)*3)+1)];
    eval(estr)
    pause
end

%% ----- Fitness-----%%
function [string1,fit,xframe,yframe] = fitness(string,numVars,option)

string1 = string;
String = reshape(string,numVars,3);
[dead_space,xframe,yframe]=dot_model(String);
fit = -dead_space;

%% -----

```

## APPENDIX B

Matlab code for Cross Entropy Method:

```
function [sol1,bestdead_space,store_result,P] = crossentropy(filename,setmod)
t = 1;
% comp_num(1:n) = 1:n;
Nol = nargin;
if Nol < 2
    [comp_num,ModDim,prop]=read_yal(filename);
else
    [comp_num,ModDim] = modprop(setmod);
end
n = comp_num;
% comp_num(1:n) = 1:n;
P = ones(n,n,4)*1/(4*n);
N = 4000;
X = zeros(N,n);
Y = zeros(N,n);
rho = 0.075;
Q_tile = round(rho*N);
Pt_hat_previous = P;
alpha = 0.75;
gamma_previous = 1000;
store_result = 100;

while (t < 5)
    for a = 1:N
        clf
        [X(a,:), Y(a,:)] = nodeplacement(P);
        state = zeros(n,1);
        phi = zeros(n,1);
        for b = 1:n
            Y_rep = Y(a,b);
            if Y_rep == 1
                state(b) = 0;
                phi(b) = 0;
            elseif Y_rep == 2
                state(b) = 0;
                phi(b) = 1;
            elseif Y_rep == 3
                state(b) = 1;
                phi(b) = 0;
            elseif Y_rep == 4
                state(b) = 1;
                phi(b) = 1;
            else
                display('error in calculation')
                t = 10;
            end
        end
        end
        X_rep = X(a,:);
        sol = [X_rep state phi];
        [xy_ctr rtt] = Decodestack(sol,ModDim);
        dead_space(a,1) = deadspace_cal(xy_ctr,rtt,ModDim);
    end
    [deadarea,idx] = sort(dead_space);
    areaknown = deadarea(1:Q_tile);
    gamma = deadarea(Q_tile);
    bestdead_space = deadarea(1);
    if gamma <= (gamma_previous+0.05) || bestdead_space < previous_deadspace
```

```

bestmodseq = X(idex(1),:);
bestposphi = Y(idex(1),:);
state1 = zeros(n,1);
phi1 = zeros(n,1);
for c = 1:n
    Y_rep = bestposphi(c);
    if Y_rep == 1
        state1(c) = 0;
        phi1(c) = 0;
    elseif Y_rep == 2
        state1(c) = 0;
        phi1(c) = 1;
    elseif Y_rep == 3
        state1(c) = 1;
        phi1(c) = 0;
    elseif Y_rep == 4
        state1(c) = 1;
        phi1(c) = 1;
    else
        display('error in calculation')
        t = 10;
    end
end
sol1 = [bestmodseq state1 phi1]
[xy_ctr rtt] = Decodestack(sol1,ModDim);
bestdead_space = deadarea(1)
LcornerLayout(xy_ctr,rtt,ModDim,bestdead_space);

Pt_hat = zeros(n,n,4);
ls_sum = 0;
for j = 1:Q_tile
    X_sequence = X(idex(j),:);
    Y_sequence = Y(idex(j),:);
    Pt = zeros(n,n,4);

    for i = 1:n
        Pty = X_sequence(i);
        Ptx = i;
        Ptz = Y_sequence(i);
        Pt(Ptx,Pty,Ptz) = 1;
    end
    S = deadarea(j);
    if S <= gamma
        ls = 1;
    else
        ls = 0;
    end
    Pt_hat = Pt*ls + Pt_hat;
    ls_sum = ls + ls_sum;
end

Pt_hat = Pt_hat/ls_sum;
Pt_hat = alpha*Pt_hat + (1-alpha)*Pt_hat_previous;
Pt_hat_previous = Pt_hat;
P = Pt_hat;
if gamma == gamma_previous
    t = t+1;
else
    t = 1;
end

gamma_previous = gamma;

if bestdead_space < store_result
    store_result = bestdead_space;
end

```

```

        str = date;
        img = getframe(gcf);
        imwrite(img.cdata, ['D:\Angel\CELcorner_result\' str filename num2str(store_result) '.jpg']);
        pause(0.5);
        sol_final = sol1;
        [xy_ctr rtt] = Decodestack(sol_final,ModDim);
        ID = sol_final(:,1);
        Frame = getframeshape(xy_ctr,rtt,ModDim);
    end

    previous_deadspace = bestdead_space;
end

end
%         [Compacted_Percent]=SimpleShift(Frame,ID,comp_num,ModDim)

```

## APPENDIX C

### Matlab Code for Random Data Generation based on Node Placement:

```
function [x, y] = nodeplacement(P)

[n,m,p] = size(P);
i = 1;
t = 0;
b = zeros(1,n);

while (t < n)
    U = rand;
    R = 0;
    Q = 0;
    S = rand;

    for j = 1:n
        for k = 1:p
            R = U*(1 - b(j))*P(i,j,k) + R; %add row
        end
    end

    sum1 = 0;
    j = 0;
    k = 1;
    while (sum1 <= R)
        j = j+1;
        if j > n
            k = k+1;
            j = 1;
        end
        if b(j) == 0
            sum1 = sum1 + P(i,j,k);
        end
    end

    P(:,j,:) = 0;

    for l = 1:n
        if sum(P(l,:)) ~= 0
            P(l,:) = P(l,+)/sum(P(l,:));
        end
    end

    t = t+1;
    i = i+1;
    x(t) = j;
    y(t) = k;
    b(j) = 1;

end
```